Universidade de Aveiro
Departamento de Electrónica, Telecomunicações e Informática,
2017

Diogo Bastos
Tavares da Silva

RTDB2: Blackboard Distribuído e Flexível

RTDB2: A Flexible Distributed Blackboard

Diogo Bastos Tavares da Silva

RTDB2: Blackboard Distribuído e Flexível

RTDB2: A Flexible Distributed Blackboard

Dissertação apresentada à Universidade de Aveiro para cumprimento dos requisitos necessários à obtenção do grau de Mestre em Engenharia de Computadores e Telemática, realizada sob a orientação científica de Doutor Artur José Carneiro Pereira, Professor Auxiliar do Departamento de Eletrónica, Telecomunicações e Informática da Universidade de Aveiro, e do Doutor José Nuno Panelas Nunes Lau, Professor Auxiliar do Departamento de Eletrónica, Telecomunicações e Informática da Universidade de Aveiro.

o júri / the jury

presidente / president

Doutor Filipe Miguel Teixeira Pereira da Silva
Professor Auxiliar da Universidade de Aveiro

vogais / examiners committee

Doutor Luís Miguel Pinho de Almeida
Professor Associado da Universidade do Porto (arguente)

Doutor Artur José Carneiro Pereira
Professor Auxiliar da Universidade de Aveiro (orientador)

agradecimentos / acknowledgements

Gostaria de começar por agradecer ao meu orientador, Professor Doutor Artur Pereira, e ao meu co-orientador, Professor Doutor Nuno Lau, por me terem proporcionado esta oportunidade, pela disponibilidade e, acima de tudo, pelo apoio dado.

Também um grande agradecimento ao Eurico Pedrosa por conceder discussões bastantes pertinentes, principalmente a nível técnico sobre o projecto, que influenciaram bastante o resultado final. Não menos importante, agradeço ao Ricardo Dias que foi partilhando os seus conhecimentos ajudando a que o projecto progredisse no caminho pretendido. Pelo apoio fornecido e sempre disponível, agradeço ao Filipe Amaral. Agradeço também ao David Simões pelos momentos de descontração proporcionados dentro do laboratório. Não esqueçendo, um agradecimento especial pelo ambiente fantástico e companheirismo fornecido pelos membros do IRISLab, estando sempre presentes e disponíveis para ajudar.

E como o projecto não é só dentro do laboratório, um grande agradecimento à Joana Gonçalves pela sua paciência e apoio fundamental.

Aos meus pais por me permitirem chegar até aqui, pela confiança depositada e, não menos importante, pelo apoio incondicional.

A todos os que estiveram presentes, um muito obrigado.

Resumo

Devido ao contínuo crescimento na área de robótica, cada vez mais existe a necessidade de robôs comunicarem entre si de modo a ser possível criar cenários de cooperação, como no caso do futebol robótico. Na CAMBADA, uma equipa de futebol robótico que participa ativamente em competições nacionais e internacionais, existe um módulo interno responsável por garantir que os robôs conseguem aceder facilmente à informação partilhada entre eles de forma simples e eficaz.
Este módulo é designado de Base de Dados de Tempo-Real (RtDB) e permite a replicação dos dados enviados por diferentes robôs, garantindo que cada robô consegue facilmente ter acesso à informação lida ou interpretada por um robô remoto. O modelo usado pela RtDB baseia-se totalmente em memória partilhada, sendo que cada robô contém a informação gerada e partilhada pelos outros replicada na sua instância. Desta forma o acesso aos dados de um outro robô é eficiente. A atualização dos dados guardados na RtDB é feita de forma transparente por um processo adicional.
O objetivo desta dissertação foi a conceção, desenvolvimento, implementação e validação de uma nova versão da RtDB, designada RtDB2, que colmatasse algumas limitações identificadas na versão anterior e simultaneamente introduzisse algumas funcionalidades novas. Uma limitação importante eliminada pela RtDB2 foi a imposição existente do conhecimento prévio do espaço em memória que um item de informação ocuparia, obrigando a dimensionar as estruturas de dados para os casos mais desfavoráveis, o que conduzia a um despedício de memória e a um custo de transmissão de informação pela rede desnecessário. Entre as novas funcionalidades introduzidas podem-se destacar a possibilidade de usar linguagens de programação diferentes para produzir e consumir o mesmo item de informação, a possibilidade de dinamicamente introduzir novos itens ou a tolerância a pequenas modificações na definição de um item.
A nova solução foi devidamente testada e utilizada em duas das competições anuais do RoboCup (Festival Nacional de Robótica em Coimbra, Portugal e no RoboCup 2017 em Nagoya, Japão), sem ter existido quaisquer ocorrências de problemas.

Abstract

Due to the continuous growth in the area of robotics, there is an increasing need for robots to communicate among them in order to create cooperation scenarios, as for example in robotic soccer. At CAMBADA, a robotic soccer team which actively participates in national and international competitions, there is an internal module responsible for ensuring that robots can easily access information shared between them in a simple and effective way.

This module is known as Real-Time Database (RtDB) and allows the replication of data sent by different robots, ensuring that each robot can easily access information that was read or interpreted by another robot. The model used by RtDB is based entirely on shared memory, with each robot containing the information generated and shared by the others replicated in its instance. This way, the access to the data of a remote robot is efficient. Updating the data stored in the RtDB is done in a transparent manner by an additional process.

The objective of this dissertation was the conception, development, implementation and validation of a new version of the RtDB, called RtDB2, that would fill some limitations identified in the previous version, and that simultaneously could introduce some new functionalities. One important limitation eliminated by the RtDB2 was the existing imposition of previous knowledge of the memory space that an information item would occupy. This would force to previously define the size of the data structures to the most unfavorable cases, which would lead to a memory waste and an unnecessary bandwidth usage. Among the new features introduced, there is the potential of using different programming languages to produce and consume the same item of information, the possibility of dynamically introducing new items or the tolerance to small modifications in the definition of an item.

The new solution was duly tested and used in two of the annual competitions of RoboCup (Portuguese Robotics Open in Coimbra, Portugal and RoboCup 2017 in Nagoya, Japan), without any occurrences of problems.

Contents

List of Figures

List of Tables

Listings

Glossary

ACID Atomicity, Consistency, Isolation and Durability.

AGPL GNU Affero General Public License.

AOF Append-only File.

API Application programming interface.

BASE Basic Availability, Soft-state and Eventual consistency.

BDB Berkeley DB.

BSON Binary JSON.

CAMBADA Cooperative Autonomous Mobile roBots with Advanced Distributed Architecture.

CAP Consistency, Availability and Partition tolerance.

CLC Cross Language Compability.

CPU Central Processsing Unit.

DBMS Database Management System.

DRDB Disk Resident Database.

FIFA The Fédération Internationale de Football Association.

FLAC Free Lossless Audio Codec.

GUI Graphical User Interface.

HTTP Hypertext Transfer Protocol.

IEETA Institute of Electronics and Informatics Engineering of Aveiro.

JSON JavaScript Object Notation.

LMDB Lightning Memory-Mapped Database Manager.

MMDB Main Memory Database.

MSL Middle Size League.

NFC Near-field Communication.

NoSQL Not Only SQL.

OSI Open Systems Interconnection.

PMan Process Manager.

RAM Random Access Memory.

RDBMS Relational Database Management System.

RtDB Real-time Database.

RtDB2 Dynamic Real-time Database.

RTS Real-Time System.

SQL Structured Query Language.

TCP Transmission Control Protocol.

UDP User Datagram Protocol.

UML Unified Modeling Language.

XML eXtensible Markup Language.

Chapter 1

Introduction

Robotics is a field of study that has gained popularity lately, and it leads to important applications that help or even replace human beings in different tasks. It can perform simple operations, such as cleaning a room, although even a simple task like that requires knowledge in different domains that deal with hardware and software. These tasks typically involve many different constraints and uncertainties, making Robotics a challenging research topic.

1.1 Motivation

Robotic soccer is one example of an application that consists of having robots instead of players playing soccer. RoboCup[1] is an international non-profit organization that was established in 1997, by hosting the first official games of robotic soccer in this project. Every year, RoboCup holds many different competitions in the field of Robotics with challenging problems, each one of them with a different domain of application: RoboCupSoccer, RoboCupRescue, Robocup@Home, RoboCupIndustrial, RoboCupJunior.

The main objective of the RoboCup is to *"By the middle of the 21st century, a team of fully autonomous humanoid robot soccer players shall win a soccer game, complying with the official rules of FIFA, against the winner of the most recent World Cup"* [1]. In RoboCupSoccer, there is a league known as glsmsl. In this league there are two teams of five fully autonomous robots playing soccer against each other, using similar rules to the ones used by The Fédération Internationale de Football Association (FIFA).

The Cooperative Autonomous Mobile roBots with Advanced Distributed Architecture[2] team, also known as CAMBADA, is one of the teams that participate in the Middle Size League (MSL). The team was founded in 2003 by the research group within Institute of Electronics and Informatics Engineering of Aveiro (IEETA) from the University of Aveiro. CAMBADA was world champion in 2008 and since then it has achieved its place on the podium.

[1]RoboCup Website: `http://www.robocup.org`
[2]CAMBADA Website: `http://robotica.ua.pt/CAMBADA`

In a soccer match between humans, the players are typically sharing information about strategies. They heavily depend on the cooperation between each other to win the matches. Also, they usually know the intentions of each other ahead, so they can move towards a strategic position. The same occurs during a robotic soccer match: the robots must share their information to coordinate with each other.

In CAMBADA, the robots use a system known as Real-time Database (RtDB) that allows them to store all the data perceived and the decisions that are made. The RtDB also allows the sharing of data with the other robots from the same team in the field. That system was developed by Frederico Santos [2][3] in 2004, with the resources and restrictions from that year. However, the technology and computer capabilities kept evolving, and nowadays there are fewer restrictions and more resources in terms of software and hardware. For this reason, the developed RtDB has to be updated with a new system that is less restrictive and use the most recent technologies to benefit the most out of it.

1.2 Objectives

The RtDB is a system based on a classical blackboard architecture [4]. This type of system is used when several sources of information are required to achieve a solution for a common problem. It allows each source to update the blackboard, so that everyone can easily see all the information that is being shared. Moreover, the RtDB is a part of the software in CAMBADA responsible for granting that all the information is shared among internal processes of the agent and the robots themselves.

The RtDB currently uses fixed-size data structures and does not apply any optimization to the data that is being sent. It is a simple solution that obtains the object from the shared memory and stores it in a preallocated space. This means that the solution is not flexible since every agent must agree on the restrictions from the preallocated space. Therefore, when new items are added to the internal structure, it must be a known modification by every agent.

This dissertation aims to improve the system previously described by providing a different approach for sharing and storing data, bringing some useful and necessary features. It can be summarized as the following steps:

1. Allow the storage of dynamic data structures – It should be possible to add dynamic structures without having the necessity of preallocating a space for it, otherwise there will be wasted space.

2. Add some flexibility between the internal storage structure of different agents – New items in one agent should not affect the behavior of the other agents (they should ignore the new items).

3. Prevent the complex logic when adding new items – The storage should allow any data structure to be added. The RtDB forces the user to modify a configuration file,

run an executable and share the result among all the agents, which is a complex step without any flexibility.

4. Improve the overall performance of the system and the bandwidth used by the system when sharing items over the network, since it is limited during MSL matches;

5. The new solution must allow to smoothly upgrade the system – It means that is must be possible to integrate the new solution without having to deprecate the RtDB at the beginning.

As shown by the objectives, the main idea of this solution is to bring improvements by benefiting the available resources and current tools.

1.3 Structure of the Dissertation

This chapter briefly describes the scope of this dissertation by introducing the importance of the communication and storage that allows cooperation in a system composed of several agents. Apart from it, this dissertation is organized into the following chapters:

Chapter 2 – Introduces the state of the art on topics related to this document. The chapter introduces the topic about databases and the most known models used, relational and non-relational. It also describes differences between using an in-memory and a disk resident database. Moreover, the chapter shows some examples and strategies used by some databases. At last, it describes briefly the concept of serialization and compression.

Chapter 3 – Presents the previous solution for this problem, known as RtDB. This chapter briefly describes how the RtDB works internally and the methods used that are relevant to this document, such as the concept of shared and local items, or even estimating the lifetime of a given item.

Chapter 4 – Describes the proposed solution, known as RtDB2, by discussing several topics. The chapter describes requirements of the system, strategies used and choices that were made. It also shows details about the implementation that was done and tools that were implemented to interact with the RtDB2.

Chapter 5 – Evaluates the state of the presented solution by comparing the performance with the previous solution and with the current deadlines. It also presents the limits of the bandwidth that were considered and compares it to the current consumption in terms of data sent over the network.

Chapter 6 – Outlines the work done with its advantages when compared to the previous solution and discusses the validation of the dissertation. It also presents some possible improvements and ideas that are left open.

Chapter 2

Data Sharing

Data sharing is a concept that can be widely understood as a practice that provides access to specific data between several entities. The act of sharing can be done in several different ways. It can be two people trying to share a file using Google Drive or it can be two autonomous machines communicating internal states about themselves, as temperatures or surroundings. Many different examples could fit in the topic of data sharing, and those cases have a large number of methods that can be used to share the data, like WiFi, Bluetooth, Near-field Communication (NFC), Ethernet, among others. Each of them has different features, such as having more range or higher transfer rate.

In WiFi and Ethernet, there are many communication protocols that can be used to share data and these protocols have foundation in specific models, such as Open Systems Interconnection (OSI) and TCP/IP model. Considering the TCP/IP model, it is a model composed of 5-layers (layer on top of another layer) from bottom to the top: Physical Layer, Data Link Layer, Network Layer, Transport Layer and Application Layer. As an example, there are protocols like Hypertext Transfer Protocol (HTTP), that stands on the top of the Application Layer and it is used to communicate through the browser or even to communicate with specific applications, like Google Drive. The HTTP is commonly used by using Transmission Control Protocol (TCP) from Transport Layer to grant a reliable connection. However, it is possible to transfer data by using other protocols, such as User Datagram Protocol (UDP) from Transport Layer. There is an endless number of methods to share the data, so the requirements are important to find out what fits best.

This document is essentially focused on a system that operates in a real-time environment. Therefore, it is required to define what is an Real-Time System (RTS) to understand what are the technologies and the requirements needed for this situation. An RTS can be loosely defined as a system that must fulfill temporal requirements; it has to act on the environment in time. This type of system typically has strict requirements regarding performance. It must guarantee a response within specified time, a deadline.

As the system evolves, the data gets more complex to be managed. Thereby it requires an efficient way to access it [5] in order to be able to process all the data in time. Otherwise, it will just miss important deadlines.

Not all aspects of data sharing are necessarily covered in this chapter, only the ones

that are required to understand it. This chapter will be more focused in key-value stores, which are a type of a database, that have the possibility to run in the main memory.

2.1 Database

In general terms, a database is one efficient approach to manage, access and update data. Typically, the data is organized conveniently to provide easy access. That can be done in a considerable amount of different approaches, commonly defined by the database itself.

Databases can be split into several different subsets according to their features. Those subsets can be defined by where the database stores its data; the method used by the database to organize the data; the possibility to replicate the data among several nodes, etc. There is a substantial variety of features that might help to define a database.

One of these subsets is defined by the method used to structure the data in the database, and there are several types that have their own technique. However, the most common methods are typically mentioned as relational and non-relational models. It is not possible to recall that one model is better than the other. Each model has a specific environment where it will fit better than the other model. Taken all together, the requirements of the system are crucial when selecting the best database model.

Some databases use an intermediate system that allows clients to connect to the database where they can perform several operations as a query or an update. This type of systems is known as Database Management System (DBMS) which is no more than a software running along with the database system. The DBMS may also allow the user to connect to more than one database. It might be useful for a situation where a single application requires the usage of more than one database. It allows the programmer to interact transparently with all of them, meaning that the programmer will most likely not notice that is handling a system with more than one database.

2.1.1 Relational database

A relational model was one of the first introduced models that gave the possibility to organize the data [6]. Along with the relational model, the relational databases started to appear. Typically, a relational database is also refereed as Structured Query Language (SQL) databases. They are known by SQL because it is the language used to query the databases. However, SQL is not the only language that may be used to query relational databases.

A relational database stores its information in tables containing rows and columns. The rows represent tuples or records. Each tuple represents an item with its respective attributes. Each column represents all values that each tuple has in a given attribute. Taken into consideration figure 2.1, the car rent will have four columns (identifier, duration, client ID and license plate) and the number of rows will be the number of entries that exist in that table. Briefly, a row contains the values that each attribute has for a given entry.

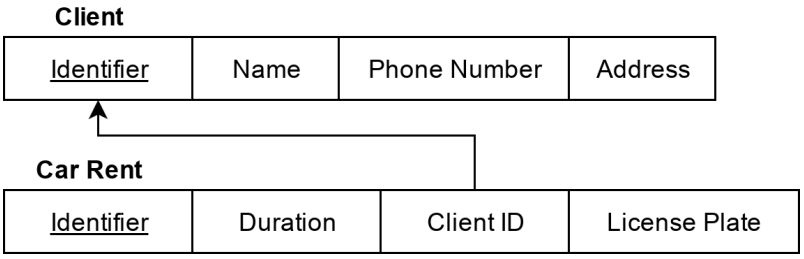


Figure 2.1: Simplified representation of a relational model with two tables. The first one is "Client" that contains tuples with 4 attributes: "Identifier", "Name", "Phone Number" and "Address". The second table is "Car Rent", and also it contains tuples with four attributes: "Identifier", "Duration", "Client ID" and "License plate". The "Client ID" (foreign key) points to the primary key in the table "Client". A foreign key is what allows the association of an object to another; a "Car rent" has a "Client" associated with it. Therefore, it would be possible to check the details of a given "Client" associated with the "Car Rent".

A relational model, as the name says, is optimized to express relations between the data. In figure 2.1, one client might have several car rents and, since the model is relational, it is possible to represent the client once and have several car rents associated with the same client identifier.

Relational Database Management System (RDBMS)

Several implementations of this model use a client-server model as shown in figure 2.2. The client uses a library in order to give some abstraction to the programmer, and it will communicate with the Relational Database Management System (RDBMS) server that will be responsible for taking actions in the database.

However, this is not the only possible scenario for this model. Some databases are serverless, which means that they do not require any server, allowing the application to interact directly with the database. SQLite is one of those examples that allows the programmer to use its Application programming interface (API) in order to interact with the database.

Transactions and ACID properties

The transaction is a concept that came as a requirement with the relational databases. It is a sequence of instructions given to the database that the programmer wants to be executed as a single operation.

A typical example is when someone wants to transfer money from its bank account to another person's account; the operation must be performed as a transaction.

Considering that this operation includes the following sequential actions:

1. The money will have to be deducted from the first person.
2. The money will be added to the second person's account.

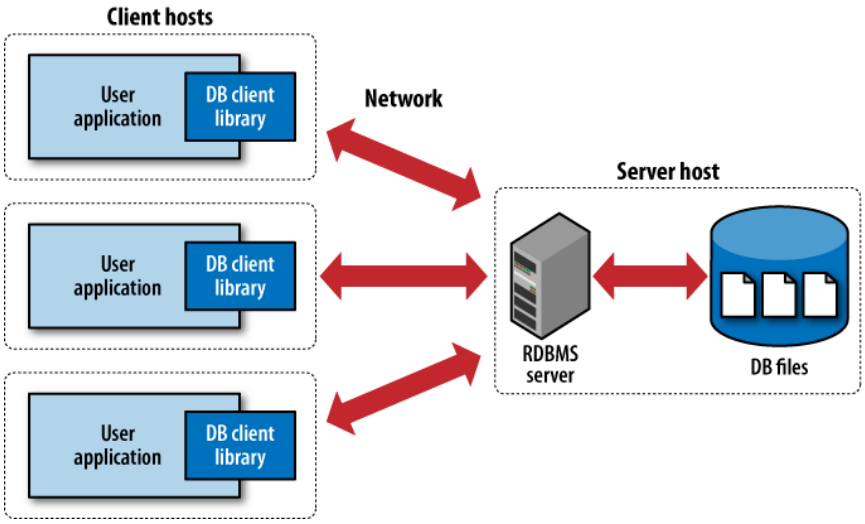


Figure 2.2: This image represents a traditional RDBMS where applications use a specific library for accessing that database, and it will communicate with a RDBMS. The RDBMS will be responsible for listening to those requests and take actions in the database - Image was taken from the book "Using SQLite" [7]

If the system, for some reason, fails during the second action, there will be a problem, because the money has already been deducted from the first person and it was not added to the second person's account, thus it has just simply disappeared. This is a clear case where transactions should be used, but not the only one.

According to Theo Haerder and Andreas Reuter [8], a transaction must have four properties to achieve the proper behavior. A transaction with appropriate behavior means that it will have all actions reflected correctly in the database, or none of them has an effect.

Those four properties, known as Atomicity, Consistency, Isolation and Durability (ACID), were thought to handle different situations:

- Atomicity - A transaction must be atomic; it must apply all operations or none of them. When part of the transaction fails, the state of the database is left unchanged.
- Consistency - A transaction must preserve the consistency of the database; modifications on the data are only allowed if they do not violate any rule that was previously defined. The database will just transition from a valid state to another valid state.
- Isolation - A transaction must be executed isolatedly from other transactions. In a concurrent system, it is important to deal with concurrency problems. In this situation, this property exists to make sure that several transactions running concurrently are not going to affect each other.
- Durability - Critical situations, such as crashes and power losses, are also important to be taken into consideration and the system must be able to recover from these sit-

uations correctly. It means that this property guarantees that after a transaction has been completed and committed, its actions will always be reflected in the database, even after a system problem.

Even though this set of properties is highly mentioned with relational databases, some databases are not yet ACID compliant. However, it does not mean that the database is not suitable for the problem. Some problems do not require the database to be ACID compliant to work correctly.

2.1.2 Non-relational database

A non-relational database stores data without explicit and structured mechanisms to link data from different buckets to one another [9]. These type of databases are mostly recognized as Not Only SQL (NoSQL) databases.

The non-relational model popularity has increased in the last few years due to the growth of the data [10] and to overcome the limitations of the relational model. Some data is so diverse that it is hard, and sometimes even impossible, to describe how it is organized. That type of data is typically mentioned as unstructured data, and it is not meant to be stored in rows and columns.

In relational databases, it is necessary to define a schema to store data. However, when dealing with non-relational databases, the schema associated with the data is much more flexible or even dynamic, since most data is unstructured, as mentioned before.

A non-relational database might deal better with high demands of data, since it has a simple model behind it. This simplifies all the logic associated with inserting new data into the database, reducing the overhead that exists related to it.

These two characteristics of the non-relational model are significant advantages when compared with a relational model. Nevertheless, the model that should be used with a system is defined according to the data that is going to be stored.

Non-relational database categories

Non-relational databases can be divided into four major categories:

1. **Key-value store** is the simplest of all the categories; it consists of having a key to access a value. This type of database is similar to how a hash table works; a unique identifier points a specific value. This value has no structure associated; thus it can be defined to be anything, giving a clear advantage in terms of flexibility.

2. **Document store** is similar to the key-value store; it has a key to access a value. However, the value is known as document, in this case, and has a specific definition. These documents are encoded in a standard data exchange format [11], such as eXtensible Markup Language (XML), JavaScript Object Notation (JSON), etc.

   A document database usually has its mechanisms to update the value, so it can easily manipulate individual parts of the document, which is encoded in some specific manner, as previously mentioned.

Key	Value
COACH_INFO	83 a4 64 61 74 65 aa 32 30 2d 30 34 2d 32 30
ROBOT_WS	de 00 16 a3 5f 69 64 b8 35 39 30 61 37 62 63 62 65 38 35 30 ...

Figure 2.3: Key-value store with 2 string keys "COACH_INFO" and "ROBOT_WS". Those keys contain binary data that was previously serialized (check section 2.4 for data serialization) and were represented in hexadecimal for this illustrative example.

3. **Column store** stores the data by column instead of rows. Therefore, a row contains all the possible values for a given attribute. The objective of switching between rows and columns is to achieve a better performance when querying the system. Storing the data in columns allows the querying of large data sets with higher performance.

4. **Graph databases** are based on the graph theory, which is a mathematical model used to interconnect nodes with edges. In this case, each node and edge can have a key and the respective value. This type of graphs is mostly used in scenarios where relations exist such as social analysis applications, dependency analysis, etc. [12] An example of this type of databases can be visualized in figure 2.4, where it is possible to view the nodes and edges with attributes in it.

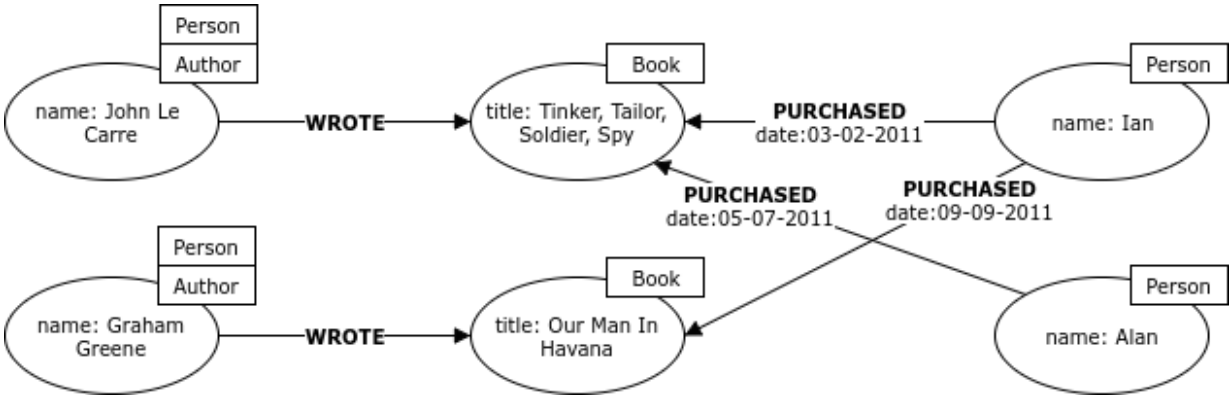


Figure 2.4: Graph database with six nodes and five edges, with their respective properties. There are two actions represented, purchasing or writing a book. Those actions are the edges between the nodes, also known as relationships. Image taken from Neo4J[1] and adapted for this example.

As mentioned before, NoSQL appeared with the growth of the data. The distributed systems started to be used to process large amounts of data from different locations. This means that keeping the data consistent became harder. Thus a new theorem appeared, known as Basic Availability, Soft-state and Eventual consistency (BASE).

BASE versus ACID

A BASE system states three distinct aspects [13]:

[1] `https://neo4j.com/developer/graph-database/`. Accessed at 4th May, 2017

- **Basic Availability** - The data store must always be available whenever there is a request. This does not guarantee that the answer will be the most recent, it only assures that there will be an answer.
- **Soft-state** - The data store does not have to be consistent at all times, tolerating some inconsistency among the replicas in a distributed system. It might occur that two instances from the system contain some differences in terms of data.
- **Eventually consistent** - The data store after some time will show consistency.

The set of properties defined as ACID have a pessimistic behavior when dealing with distributed systems; it means that ACID properties assume the worst case when handling with distributed systems, which might not be the best method to evaluate them.

BASE system is the opposite of ACID. It is more focused on the availability of data allowing the system to stale some data. On the other hand ACID is more focused on the consistency.

CAP

An important theorem, Consistency, Availability and Partition tolerance (CAP), has emerged from Eric A. Brewer [14] to explain the existing trade-off between availability and consistency in distributed databases.

The acronym CAP denotes the following three characteristics:

1. **Consistency** means that the data obtained is always the most recent;
2. **Availability** implies that a system has always a response even if it is not the most recent one;
3. **Partition tolerance** emphasize that the system must keep working even when there are some failures that cause a node to go down.

The **Consistency** sometimes can be confused with the one defined by ACID, although they are not related at all. In figure 2.5, it is possible to verify a situation where the data obtained is not always the most recent one. Therefore, the consistency from the CAP theorem is lost.

In the example of figure 2.5, it is possible to find three replicas ("Leader", "Follower 1", and "Follower 2") and three clients ("Referee", "Alice", and "Bob"). The "Referee" announces the final results of a football match by inserting it into a replication known as "Leader", however, "Follower 1" and "Follower 2" have not received it yet. After "Leader" replicates it to the "Follower 1", "Alice" is already able to see the final results of the match, but "Bob" still thinks that the game is not finished yet because he has not received the update about the final scores. Consequently, this system is not consistent; the replications must grant that every client sees the same state.

According to the CAP theorem, a distributed system can only **have at most two of these properties**. Thus, it is impossible to achieve the three properties at the same time.

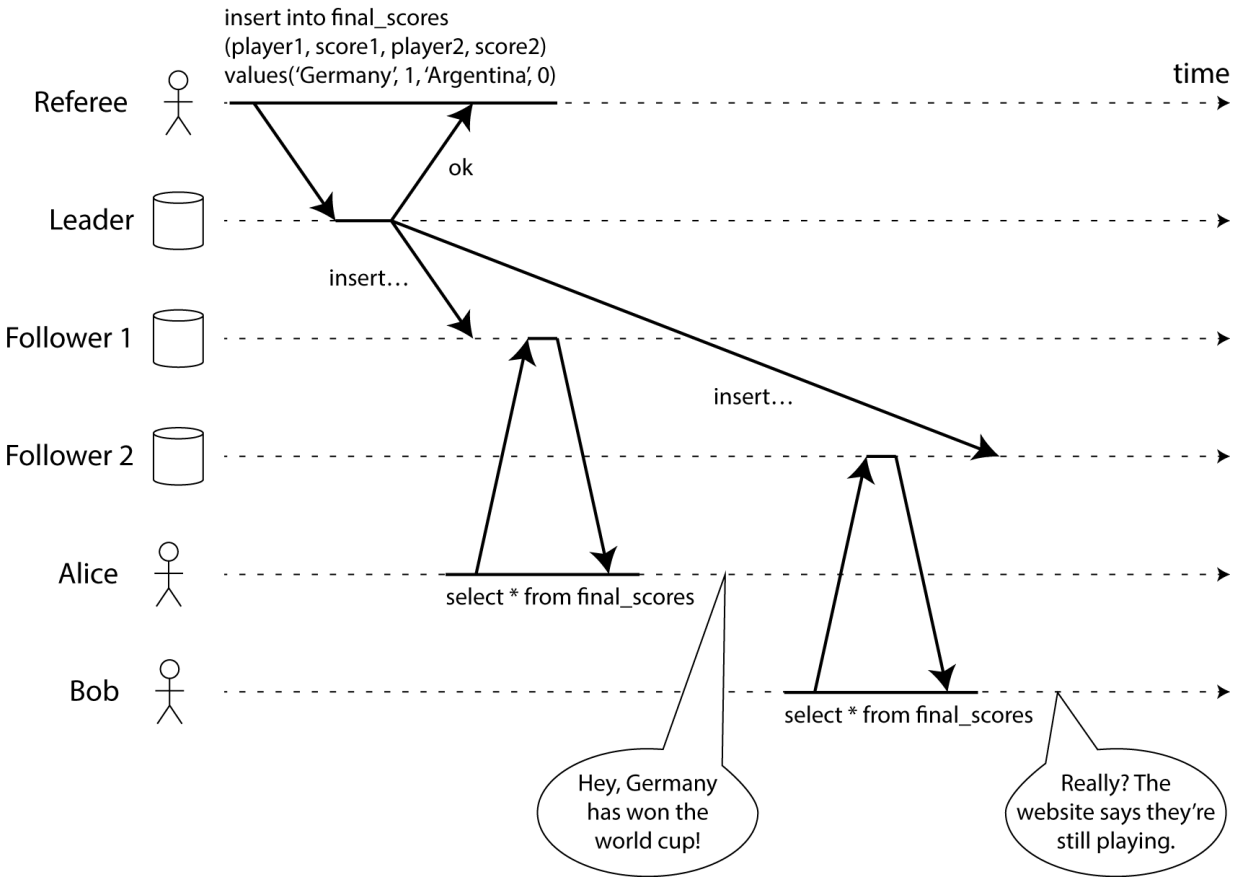


Figure 2.5: This figure represents a system that is able to replicate its storage. The replications are known as "Leader", "Follower 1", and "Follower 2" and the clients of those replications are known as "Referee", "Alice", and "Bob". This diagram represents the state of the system viewed by every client at a given time. The figure shows the insertion made by the Referee being replicated by the system and retrieved by the other clients. Image taken from an external source[2]

Usually a distributed system is CP or AP. A CP grants consistency over availability where it will always respond with the most recent answer. And a AP gives priority to availability by discarding consistency; it means that it will always answer to the request, however it might not be the most recent answer.

As shown by the theorem, it is said to be impossible to achieve the three properties at the same time. Therefore, a system must have well-defined requirements to understand which one of the three characteristics can be discarded.

[2] `https://martin.kleppmann.com/2015/05/11/please-stop-calling-databases-cp-or-ap.html`

2.2 Main memory versus disk resident Databases

Despite the model that a database is using, there are more aspects to be taken into consideration. Databases can be classified according to where they mainly store the information, having two opposite categories: store the data on the disk or the main memory.

A databases that stores data on disk is known as Disk Resident Database (DRDB), and as a result, it will be limited by the response time of the disk, although it will also benefit from the disk space, which is typically bigger than the main memory size. Even though there are methods that help mitigate that limitation related to the response timing like caching.

On the other hand, the database information can also be stored in memory, also known as in-memory database or Main Memory Database (MMDB). This type of databases strongly benefits from the response time of the main memory. However, it also has some drawbacks with it. The main downside of MMDB is that the database is limited to the main memory size that is usually small compared to the disk space. Another drawback is that the main memory is volatile, meaning that the memory's content is lost when the power is turned off or interrupted [15].

There are drawbacks in both types as mentioned before, although every database architecture has methods to minimize those problems.

Disk Resident Database (DRDB) caching

A general solution that most DRDB use to mitigate the problem related with the response time in the disk is using a cache. The cache's job is to keep the most used data in the main memory to benefit from its response time. Even considering that the whole data in the database is cached in the main memory, the system can not completely benefit from it. The internal mechanisms of the database would still believe that the data is mainly stored on the disk, such as indexes structures from database that were designed for disk access [15].

Another solution that could appear is to use the DRDB stored in a Random Access Memory (RAM) filesystem. This solution would use the memory for all data instead of part of it, like the case of the cache, although the internal mechanisms to access the data would be optimized for the disk as well. In both solutions, it is not possible to entirely benefit from the main memory without having some overhead.

Typical problems in a Main Memory Database (MMDB)

In scenarios where the database is an MMDB, there are drawbacks, as mentioned before; some inevitable and others that can be mitigated. The MMDB uses the main memory causing the data to always be limited by its size, and that is unavoidable. Thus the only possible solution is to guarantee that the data does not go beyond the limits of the main memory.

The other problem about MMDB is that the main memory is volatile. In some scenarios, the database does not need to be persistent, and the data can disappear after a power loss of the system. However, some scenarios require the data to be persistent. One solution to this problem is to use a backup copy of the database on disk [15], in order to prevent loss of information when the system crashes or turns off.

In summary, it is not possible to conclude that one is better than the other, it will always depend on the requirements of the system. It might be better to use MMDB in a system that requires as much performance as possible and does not require a non-volatile storage. An example of a use case for the MMDB is a system that uses database to share information among internal processes and does not require to keep the data for long periods of time.

2.3 Database Candidates

There are many different options when selecting the right database system. Over the years, a large number of databases have been developed in order to fulfill the requirements of each system.

The system described in this document (later on chapter 4) updates and retrieves data very frequently. That retrieval and update typically involves the whole database and not only some fields, therefore it is a high demanding system in terms of performance. It uses the database as a repository to share the information among internal processes, meaning that it is not important to keep the information after a shutdown. Only in-memory databases were considered because they tend to achieve better response times than disk resident databases (see section 2.2 explaining the difference between in-memory and disk-resident systems). Moreover, this system does not require the stored information to be retained when the system is shut off (check section 4.1 for more details about the requirements of the system).

Table 2.1 shows that key-value stores can achieve higher performance and scalability, and yet still be able to benefit of a high degree of flexibility. Key-value store is a simple model that allows to insert or retrieve a value on a specific key and does not have any additional functionality. Even though that document stores and graph databases have a high degree of flexibility; they were not considered at the beginning since they offer unnecessary complexity.

Considering the existence of these categories, only key-value store databases were considered as possible candidates for the system described in this document due to the fact that simplicity will lead to speed [17] and a simple key-value store is enough to comply with the requirements (section 4.1).

The relational model was not considered because it would have an overhead associated with the relational logic itself, which is not required for this type of system. In the relational model perspective, this system would fit in a single table with two columns, key and value. There is no relation between the data in order to use this type of databases and there are no benefits in using a single table over a key-value store. Typically, NoSQL databases are

Table 2.1: Comparison among several database models (key-value, column, document, graph and relational) in terms of performance, scalability, flexibility, complexity and functionality. These metrics are not specific for any database implementation, so it is only possible to get a general understanding of how each model should behave. This table was taken from "NoSQL Databases" [16].

Type	**Performance**	**Scalability**	**Flexibility**	**Complexity**	**Functionality**
Key-Value stores	high	high	high	none	variable (none)
Column stores	high	high	moderate	low	minimal
Document stores	high	variable (high)	high	low	variable (low)
Graph databases	variable	variable	high	high	graph theory
Relational databases	variable	variable	low	moderate	relational algebra

faster than relational databases [18][19].

Mainly column stores, document stores, and graph databases were also excluded for one of the reasons that relational model was also eliminated. It is unnecessary to introduce more features to the database when they are not going to be used with the system.

The following alternatives always offer the possibility to store the data in the main memory, and they also may work as a key-value store.

2.3.1 Non-distributed Databases

A non-distributed database means that the whole data is available in a single location. Nevertheless, it is possible to implement a distributed system by using several non-distributed databases. The following examples of non-distributed databases are also embedded databases, meaning that they are highly integrated with the system and the database might not be seen as an external resource to the system. Only the most important databases were mentioned here even though most of the well-known possibilities were considered.

Berkeley DB

Berkeley DB (BDB)[3] is a high-performance embedded database and it is mainly known as a data store for key-value data. It is a library with a huge API that allows the configuration of the database in numerous different manners, adapting it to the situation where it is going to be used. Even though it can be configured as a distributed database by enabling the replication, it was mostly analyzed in this document as a non-distributed database, discarding the possibility of using the replication API.

One of the possible configurations allows to choose where the data is stored. It can be stored in the disk, or it can use the main memory in order to achieve better performance

[3]Oracle Berkeley DB 12c overview page: `http://www.oracle.com/technetwork/database/database-technologies/berkeleydb/overview/index.html`

and use the disk for backing up the data (as explained in section 2.2). BDB allows to be used as a cache, but besides that, it also allows to use memory-mapped files.

Memory-mapped files have their data directly mapped into the memory, meaning that the access is much faster. An example of how a simple situation of memory-mapping works is represented in figure 2.6, where it has part of the file directly mapped into the process's address space. In the case of the BDB, it only allows to use memory-mapped files in a read-only database. However, it will operate on records stored directly on the memory without any overhead from cache management [20].

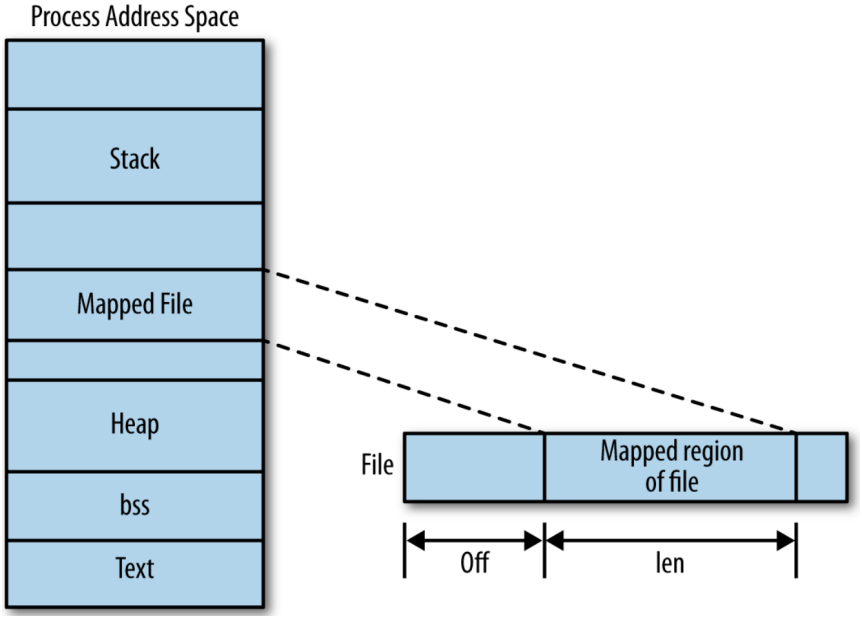


Figure 2.6: This figure represents a file that is memory-mapped directly into the process's address space. It means that when the mapped chunk is accessed, it is actually being accessed in the address space, and not on the disk. It also shows a given offset on the file that corresponds to the possibility given by an argument in the Linux call `mmap()`. It allows to map the file only after a given offset. This figure was taken from *Linux System Programming* [21]

Another advantage of memory-mapped files is that they are not limited by the main memory size. Their size is limited by the disk space and the size of the address space. Moreover, the files are directly mapped into the memory, but the operating system is responsible for moving the data as needed, in and out from the available memory.

BDB has transaction semantics in its API with full supported ACID properties. These transactions grant atomicity, consistency, isolation, and durability (more details about ACID in section 2.1.1). In order to support ACID properties, it had to consider several mechanisms such as writing logging before writing data pages. It is also known as write-ahead logging [22] and it gives further support to implement a system reliable to crashes in case of data corruption or loss [20].

Besides features that were implemented to support ACID properties, there are some important characteristics that affect a system where it is going to be used. This library

can work in an environment with multiple processes, and it also supports multi-threaded environment by the use of certain flags when opening the database.

The locking mechanics of the BDB allows multiple readers and a single writer, meaning that a reader will not cause another reader to block. A writer will be blocked once it tries to write anything with read operations in progress. An important detail to notice is that both, readers and writers, have locks associated with them, which have some overhead associated.

BDB API is written in `C` with many API bindings, supporting several languages rather than only `C`, such as `C++`, `Java`, `Perl`, `C#`, `Python`, and many others. This database is open-source under a GNU Affero General Public License (AGPL).

LMDB

Lightning Memory-Mapped Database Manager (LMDB)[4] is a database that was developed by Symas for the Symas OpenLDAP project. It is an extremely compact database that has a small footprint. Besides being compact, it is also known for having a good performance. This database does not include any other features that are not necessary in a simple key-value store. This database has small footprint due to its simplicity.

The database has full ACID semantics. It is impossible to operate over the database without using a transaction; consequently, everything is a transaction. This database allows multiple readers that do not block each other and a single-writer. However, the readers are not blocked by the writers and writers do not block the readers as well. The only thing that exists is a mutex to prevent multiple writers at the same time [23]. This result is achieved by using a copy-on-write semantics as shown on figure 2.7.

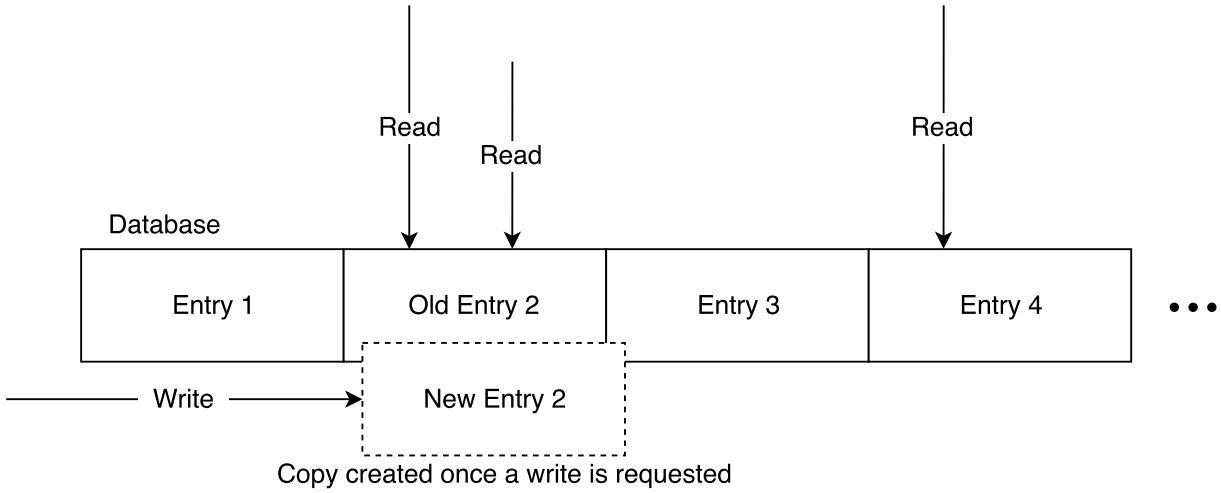


Figure 2.7: This example represents a copy-on-write semantics on a database with several entries. A write operation copies the entry in the database to write its value on it; consequently, every read operation can still obtain its values with a write operation in progress.

[4]Symas LMDB page: `https://symas.com/lmdb/`

The copy-on-write semantics says that when a write operation occurs, it must copy the entry and then write on the entry that was created. Since the entry was copied, the older entry must stay in the database until the write operation is completely over. This way the readers can still use the old entry to read its value and the writer can use the new entry. This method has several advantages, such as not requiring locks for a read operation, or granting consistency when something goes wrong during a write operation. However, the result from the write operation might be lost during a not committed operation before the crash.

Furthermore, this database uses memory-mapped files in order to achieve a better performance similar to BDB, although the strategy is completely different: it does not use any caching on its back-end. As a result, it prevents data from being replicated where it is not necessary. As an example, with a single cache system, the data can be replicated in the cache and in the main storage [23]. Moreover, it uses an approach known as *Single-level Store* that helps to create this desired behavior. This approach handles the memory as a single-address space and the operating system is completely responsible for dealing with the database needs in terms of caching. The operating system will be paging the data into main memory as needed, so that only portions of the database are actually read into the memory [24]. The pointers returned by the API are directly pointing to the keys and their values in the memory address, avoiding to copy any data. It also means that zero-copy is supported and these pointers must be taken with care.

LMDB's API is similar to the one used by BDB, although it contains lesser available functions. It is available in several different programming languages, such as `C`, `C++`, `Java`, `Python`, `Rust`, `Perl`, and others. LMDB is an open-source library under a OpenLDAP License that is a permissive non-copyleft free software license, similar to MIT License. This means that it is possible to create a derivative work to apply small modifications on it under other license.

2.3.2 Distributed Databases

This type of databases is typically defined as being a logically interrelated collection of shared data which is physically distributed across database nodes or instances over a computer network [25].

Many methods allow the sharing of data with other instances. Generally speaking, there is replication and partitioning. The method replication consists of copying the dataset to each existing instance of the database; this is also known as fully replicated. Partitioning consists of dividing the dataset into shards (or fragments) and attribute each one of them to different instances, thus the data will not be repeated in any instance. There is also a concept known as partial replication, which is a middle point between full replication and partitioning, meaning that the dataset will be replicated over the instance, but not completely [26].

Redis

Redis[5] is one of the most popular NoSQL databases. Redis is known as a data structure server that stores its data in the main memory. Even though Redis is most of the times mentioned as a key-value store, it is not just a store since it works as a server [27]. It also allows the storage of string values like most key-value stores and other types. These other types include lists, sets, sorted sets, hashes, bitmaps, and HyperLogLogs. It allows to manipulate them directly using their API without having to remove a serialized value (check section 2.4 about data serialization) from the database, modify and reinsert it.

This database is a versatile and flexible system concerning configuration. It allows the adaptation of many aspects to fit the data that is going to be stored more appropriately. It even implements a publisher/subscriber messaging, which is one of the reasons why this database is so popular. That paradigm consists of a system composed of channels, where each channel might be subscribed by a client interested in that particular type of data. When a publisher sends data to that channel, the subscribers that have revealed an interest in it will be instantly notified about the new message sent by the publisher to the channel.

As mentioned before, Redis is an in-memory database and usually a system that runs in the main memory will not be able to recover its data after a restart or a power outage. However, Redis provides two different methods in order to create some persistence, allowing to benefit from its in-memory performance and still have the possibility to restore its records after a power loss. Those two methods are:

- RDB method is a typical method used to create some persistence when using in-memory databases. It consists of taking snapshots of the dataset at specific moments. This approach allows the setting of a pre-condition when creating the snapshot based on the number of changes, allowing to snapshot only when there are a significant number of changes.

- Append-only File (AOF) method consists in logging every operation successfully made by the server. When the system restarts, it will replay all the operations in the logging; it will take more time to reboot than RDB. However, it also allows defining when the data from this method is flushed to the disk, which makes the system more durable. The result is that the changes are more likely to survive to power loss in this method.

Redis supports replication with a master/slave setup. It replicates by having copies of the master's dataset into the slaves. Consequently, master will have to replicate its dataset to the slaves since it is the only one accepting writes [28]. Redis also has the possibility to write on slaves, although those writes will be lost after a slave restart or a masters re-synchronize with the slave. This database allows several different setups using the configuration in order to replicate. For example, if a master loses the connections with the slaves, it is possible to promote a slave to master. This replication is always asynchronous,

[5]Redis Website: `https://redis.io/`

meaning that the master will not update slaves with changes when it receives a write, but only after a specific period of time.

Besides full replication, it is also possible to partitioning in order to allow larger databases to sum up the main memory of several computers. This enables the fitting of the whole data or even the achievement of a greater computational power.

2.4 Data Serialization

The term data serialization is generally understood as the act of translating a data object or a data structure into a well-known format. The reverse process is also known as deserialization and it allows to recover the data from a specific format that was used to serialize it. These processes might be required in different scenarios, such as storing a data object in a database, or transferring a data structure over the network.

A typical scenario where serialization might be required is when copying data over the network. This type of scenario is solved many times in the programming language `C++`, using a function called `memcpy`, that allows the copy of a block from the memory directly to a specific destination. However, this method does not convert any object into a known format, it is just copying the bytes representation of an object into another location. This solution might seem to work in simple scenarios where there are only structures with primitive types and without any pointers, but it can also fail when the system architecture between the machines differs, such as big-endian to little-endian representation. Otherwise, it will most likely fail to read the data properly when trying to retrieve back the data.

There are many serialization libraries that allow the application of the process described without copying directly the object representation in memory but its values, which is the important data to be transferred.

It is important to check the capabilities of each library and compare them to pick the best option. These libraries might vary in several aspects when they serialize the data, such as:

- The output's format of the serialization; it might be readable text or it might not even be human-readable. Some serialization library might include the option to switch between a readable text or a binary output.

- Mandatory usage of a schema; Some serializers require the usage of a schema that previously describes the object that is going to be serialized in a specific language. Figure 2.8 shows an example of one schema that is compiled and converted into a defined programming language.

- Cross compatibility between programming languages; it might be important that the serialization is able to handle an object in different languages. It means that it is able to serialize in a language and deserialize in a different one.

- Schema evolution; how strict the serialization library is when deserializing an object that has changed since it was serialized. Schema evolution might be useful when there

```
message TweetFeed {
  repeated Tweet feed = 1;
}

message Tweet {
  int32 id = 1;
  string title = 2;
  string message = 3;
  repeated string tags = 4;
}
```

Figure 2.8: Example of an object representation in the serialization library Protocol Buffers language[6]. This small sample of its language represents a list of tweets and each tweet contains an identifier, a title, a message and a list of tags

---

are changes occurring on the schema frequently. It is important that the library is able to handle them and still be able to deserialize as much information as possible, even with the changes to the schema. Those changes might include renaming a field, changing the field's type, removing or adding a field, etc.

There are more differences that a serializer might have, but these were considered the most important due to the fact that allow the system to benefit a high degree of flexibility when the serializer supports them. Some serializers even apply some compression to the data (see section 2.5 about compression). However, it might not be important in some problems. The choice of the serializer must be adapted to the problem.

2.5 Data Compression

Data compression is a term associated with the primary objective of minimizing the space required to represent some data [29]. The format of the original data is changed, allowing it to be represented with a lesser amount of bytes. This technique is generally used when it is required to reduce the original size of the information with different objectives, such as optimize the data's storage, reduce the network bandwidth used, improve the speed in which a file is transferred, etc. This term can be subdivided into two categories, lossless and lossy compression [30].

A lossless compression consists in compressing the data without losing any data during the process, meaning that it is reversible and it is possible to get the original data.

On the other hand, lossy compression consists in compressing the data with the possibility of discarding some information about it. This might not even be noticed by the person who is visualizing the data. However, this process is not reversible as lossless compression is. Lossy compression is mostly used in scenarios where it is possible to lose some

[6]Protocol Buffers library: `https://github.com/google/protobuf`

information and keep representing it. Those scenarios are essentially multimedia files where it is possible to lose some of the multimedia quality and keep showing it.

It is important to always consider not only the ratio by which the data got reduced but also the resources the compressor requires when applying compression and decompression. In this document, it is being considering compression of data, meaning that none of the data can be discarded, otherwise it might get corrupted and unreadable. Therefore, the compression described throughout this document is always lossless.

In terms of lossless compression, there are many algorithms already implemented for a specific purpose, such as Free Lossless Audio Codec (FLAC) that is used for audio formats. However, there are some of these algorithms that can be used for general purpose, instead of being specific for audio, video, etc. One of the most famous general purpose algorithms used for compression is known as Lempel-Ziv [31] and there are many compressors that implement it, such as `lz4`, `snappy`, `zstd`, etc.

Chapter 3

Real-time Database

Many robotic applications that require coordination of several autonomous mobile robotic agents to achieve a common goal will always have the necessity to share and store information. This chapter presents an already existing solution [2][3] for this problem that allows the data to be exchanged between mobile robotic agents.

This solution, known as Real-time Database (RtDB), is used in a RoboCup Middle-Size robotic soccer league by several teams, such as the team CAMBADA (check section 3.1 for more information about the robotic soccer team), the Tech United from the Eindhoven University of Technology and others. This solution can also be applied to different scenarios where it is important to share data between external agents or even internal processes of a system.

In the RoboCup Middle-Size robotic soccer league, it is important that the robots move fast with accurate trajectories, but it is also important that they share their information with the other robots frequently. Moreover, each robot should obtain a perception of the other robots surroundings as soon as possible, allowing them to decide their next action based on the information received through the network. This type of situation are affected by real-time constraints that should be taken into consideration.

This chapter covers briefly all the relevant aspects about RtDB, giving more focus to the ones important for this document. The RtDB is similar to a distributed database; it replicates its data to other existent instances of the database. This way every instance of the system will perceive the data received from the other robots as local data, preventing delays from requesting data only when needed. It also provides an abstraction to the user: the programmer does not need to know how the replication works to obtain data from other robots. There is no difference when accessing an item[1] from the robot itself or from another robot, this is explained later in this chapter.

[1] An item consists of a key and its respective value stored in the RtDB. The value can be retrieved or inserted through the key.

3.1 CAMBADA

CAMBADA acronym stands for Cooperative Autonomous Mobile roBots with Advanced Distributed Architecture (CAMBADA)[2]. It is a RoboCup Middle Size League (MSL) soccer team that started officially in October 2003 at the University of Aveiro.

CAMBADA team, as mentioned before, participates in the Middle Size League, which is one of the categories of the RoboCup Soccer League. The teams in this league play with five fully autonomous robots with a regular size FIFA soccer ball. Those robots are built by each team with their chosen hardware and software. The hardware has some restrictions in terms of size and weight that are imposed to the robots in order to assure a fair game. This league is played according with FIFA rules, although those rules are slightly modified for the robot players.

So far, CAMBADA has achieved remarkable positions in several competitions since the team has started. CAMBADA has participated in national and international competitions: RoboCup world championships, European RoboLudens, German Open, Dutch Open and Portuguese Robotics Open, known as Robótica.

3.1.1 Architecture

CAMBADA has adopted a software architecture where the RtDB acts as a middleware that supports the sharing of data among agents (robots) and among processes in the same agent, as shown on figure 3.1. The figure shows that each agent has several internal processes that store and retrieve information from the RtDB. There is also a special process responsible for the data replication, known as communication manager (`comm`). This process collects the data from the local RtDB storage where it is running and sends it over via wireless. Besides that purpose, it is also reading from wireless in order to store the information sent by the other running agents.

3.2 Data storage

The strategy adopted by RtDB is to use shared data region among several entities that are able to read and write data on it in order to solve a common problem. This type of systems is known as Blackboard [4], and it is a general solution that allows data to be shared without having to deal with data flows to share the information correctly. Blackboard systems started to emerge in the 1970s in order to overcome problems related with signal-interpretation, as for example Hearsay-II [32].

In a general concept, a Blackboard does not require to know where the data goes next in order to be shared. It is a communication medium and buffer that allows every module

[2]CAMBADA website: http://robotica.ua.pt/CAMBADA/

[3]Software Structure document can be found under Competitions at CAMBADA website: `http://robotica.ua.pt/CAMBADA`.

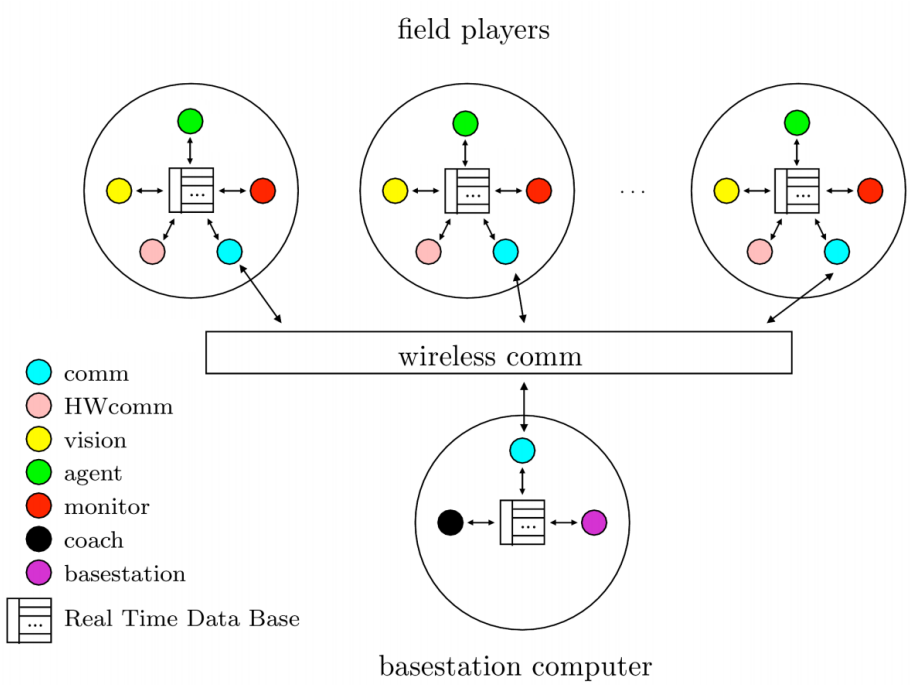


Figure 3.1: This figure represents the software architecture from CAMBADA and how the data is shared across all players and the basestation. In this example, there are 3 field players and a basestation communicating using wireless. Each field player has several internal processes represented with circles communicating with the RtDB. Image taken from CAMBADA website in the Software Structure document[3].

---

to obtain anonymously specific data that it desires [33]. It also allows the system to be flexible since every process is able to manage the data as it wants.

RtDB uses shared memory to store its data. It is organized in a strict way, meaning that all the memory is reserved when the first process in that agent initializes. This means that every data that is going to be stored must be well-known; its data size must be known. The fact that this memory reservation occurs on the start is a disadvantage, because it does not allow to have dynamic data structures as linked-lists, since it is impossible to know the size of such structures. To solve this problem, the RtDB assumes a maximum size for such structures and allocates it at the beginning. Consequently, the size of the RtDB does never change since it is allocated on the start and there might be bytes that are never used.

3.2.1 Internal Storage

This system stores the data internally into two different areas in order to allow replicating its data easier:

- **Local** area is typically an agglomerate of data items that are not required to be perceived by other agents, being only relevant to local processes. This is useful when

the data item is useless for the other agents or when data that is too large to be sent over the network.

- **Shared** area is similar to the local area, but instead of having the local items, it contains items that are going to be shared among all agents. In fact, each robot will be storing the data received from other robots. There is a process responsible for handling this area in order to replicate the information with all the agents, as described in section 3.4.

The internal structure of the RtDB can be visualized in figure 3.2. Internal storage contains a set of areas divided into two groups. One of them corresponds to the local data of the agent and the other one corresponds to the data each agent shares with the others. When referring to the shared area, it is composed of several slots (equal to the number of robots): in one of the slots, the one corresponding to the agent's identifier, its contents is generated locally; in the others, it is received from the network. Each area is composed of a list of `TRec` items and the data itself, as shown on figure 3.2.

A `TRec struct` is responsible for storing the information that describes the data itself, just like metadata. That structure has a unique identifier, an offset to the data pointing the beginning of that area, the size of the data, a timestamp field to calculate the age of the item when it is retrieved, a period field that indicates the periodic interval in cycles when an item should be shared or not and `read_bank` that flags the correct data buffer to read.

Each data item is composed of two buffers, meaning that each item will have its data replicated on the buffers. Those two buffers are used to solve the synchronization access of the processes to the data [3], meaning that the `read_bank` points to the buffer that is safe to read and the other buffer is used to write the data on it, if needed. When the buffer that is available to be written has received new data, the `read_bank` switches to it, allowing every reader process to obtain the most updated data of that item. This mechanism will works in an environment where there is at most one writer for each data item, although there can be several readers accessing the data.

3.2.2 Configuration

The RtDB requires a configuration file that specifies how many agents there are and the size of each data item. This configuration file is generated through two steps:

1. The first step corresponds to the creation of a file that expresses the configuration of the data that is going to be stored into the RtDB, usually known as `rtdb.conf`. That file contains an enumeration of the existing agents, a list of the possible data items, several schemas that allow to define which items will be local or shared, and the assignment of schemas to agents. Each data item is identified with its *class* or *struct* name and *header file* location.

2. After creating `rtdb.conf`, the `xrtdb` application must be executed. This tool (`xrtdb`) is responsible for translating the `rtdb.conf` file into other files suitable for RtDB

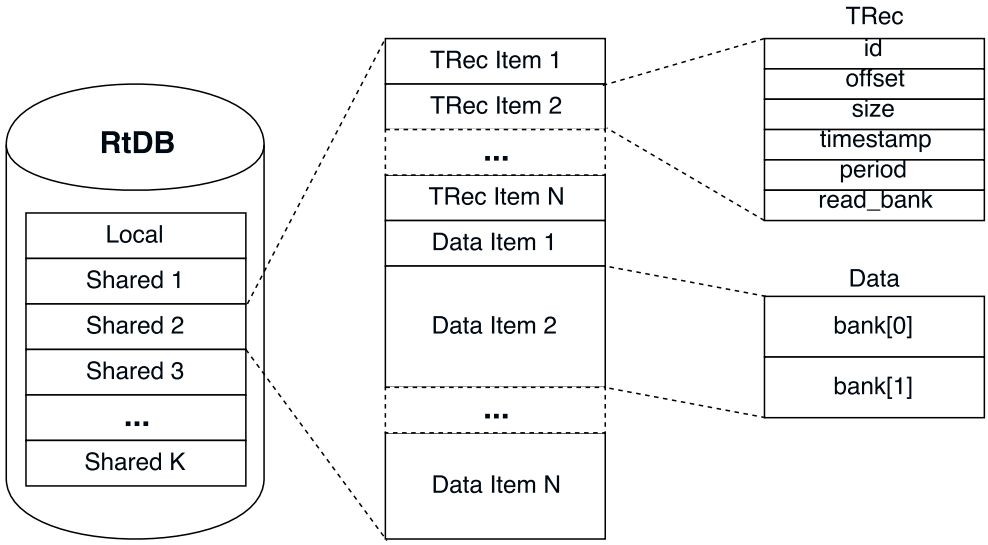


Figure 3.2: Internal storage of the RtDB - It consists in a set of areas, several shared and one local. Each area consists in a continuous set of `TRec`, one for each item, that is followed by the data, and each `TRec` structure contains several fields: `id`, `offset`, `size`, `timestamp`, `period` and `read_bank`. Each data item after the sequence of `TRec` sequences contains two copies of the data used for double buffering [3].

---

initialization and usage. The files created by `xrtdb` are: `rtdb.ini` that contains a list of data items for each agent, being each data item composed by its identifier, size, periodicity and if it is shared or local. `rtdb_user.h` that is a *header file* containing a list of `C`/`C++` macros that convert each data item name to their integer identifier.

The files mentioned above are shown in appendix A along with a brief description.

3.3 Application programming interface

The API of RtDB was developed in `C` and it is composed of a set of twelve methods. Five of them are to be used by normal processes:

1. `int DB_init(void)`
   Initializes the storage, as described before in section 3.2.1, accordingly to the configuration file. That initialization is associated with an agent identifier that will be used to distinguish from the other RtDB instances. That identifier must be defined under an environment variable with the name "AGENT".

2. `void DB_free(void)`
   This function is responsible for destroying the segment allocated in the shared memory, and should be invoked once there is nothing else required from the RtDB.

3. `int Whoami(void)`
   Returns the environment variable value set on the initialization.

4. `int DB_get(int _from_agent, int _id, void *_value)`
   Allows to retrieve a data item from the area of a given agent. On success, it returns a positive value, corresponding to the age of item. On error, it returns a negative value, corresponding to an error code.

5. `int DB_put(int _id, void *_value)`
   Inserts an item, local or shared, into the RtDB. An agent can only insert data into its own area.

The communication manager, responsible for the replication of items among the different agents, needs to be initialized and configured. Moreover, it needs to put items into the local other agents' slots. The next two functions are to be used by the communication manager:

1. `int DB_comm_ini(RTDBconf_var *rec)`
   Initializes the communication process and loads the configuration file.

2. `int DB_comm_put (int _agent, int _id, int _size, void *_value, int life)`
   Inserts a data item that was shared by another agent. This function is used to create a local replication of the data item for faster access.

The CAMBADA project developed a simulation environment that allows to test all the software without using the real robots. When this simulation is used, the high level software of all the agents runs in the same computer. Thus, the storage areas of the RtDB of every agent lies in the same computer as well. Five more functions exist to support this situation:

1. `int DB_init_all(int _second_rtdb)`
   Initializes all RtDB instances that are going to represent the agents on the simulator.

2. `void DB_free_all(int _second_rtdb)`
   Frees the memory after the execution.

3. `int DB_get_from(int _agent, int _from_agent, int _id, void *_value)`
   Allows to retrieve data from any area of a RtDB instance.

4. `int DB_put_in(int _agent, int _to_agent, int _id, void *_value, int life)`
   Allows to insert data into an area of a RtDB instance.

5. `void DB_set_config_file(const char* cf)`
   Allows to specify a custom location for the RtDB configuration file.

3.4 Data replication

The data replication consists into sending the data that an agent has stored to another instance of the system. It is done using an additional process in RtDB that is responsible for sending and receiving data from the network, as shown in figure 3.3 under the name RtDB Communication Manager, which corresponds to the `comm` process in figure 3.1.

That communication manager has two threads:

1. A sender thread that is responsible for sending the data over the network. In the RtDB not all the data that it stored is going to be replicated, only the one generated locally and labeled as shared. As an example, on agent 2, the RtDB Communication Manager will only send over the network the shared data items related with its identifier, in this case, it will only pick the shared data items from agent 2, as described in section 3.2.1. The other shared items are used to store the information received from the other agents.

2. A receiver thread that is responsible for receiving data that is sent by the other agents and storing it in the corresponding areas. As an example, agent 2 will be waiting for data from agent 1 and agent 3 to store into the local replicas (shared 1 and shared 3 regions of agent 2 RtDB, respectively).

The RtDB memory is completely allocated at the beginning with the structure shown in figure 3.2 from section 3.2.1 and as show in this section, the data inside that structure will be replicated to every agent. The user does not need to have any concerns about this replication, it is done without any explicit instruction. However, the replication has specific requirements. The data structures between different machines must match each other, it means that no fields can be swapped or missing. Otherwise, one of them will fail to copy the data when receives the data, since it does not have the expected format. This limitation of having small changes (such as, missing field or renamed field) in a structure between two data structures is referred as drifts in the data structures.

There are other factors that might affect the format of a given structure between machines, such as the architecture being used. As an example, one system has a little-endian representation in memory and the other one has a big-endian representation, it will cause the format to not be compatible when replicating the structures, although the system still works correctly between the local processes.

3.5 Data item lifetime

This system also takes in consideration the lifetime of each data item, meaning that it is possible to know how old a specific data item is. As an example, a process stores a data item and another process retrieves it 600 milliseconds later; the latter has the information that the item was stored in the RtDB for 600 milliseconds. This lifetime allows the agent to discard a data item when it is too old, or even to discover when an agent has crashed.

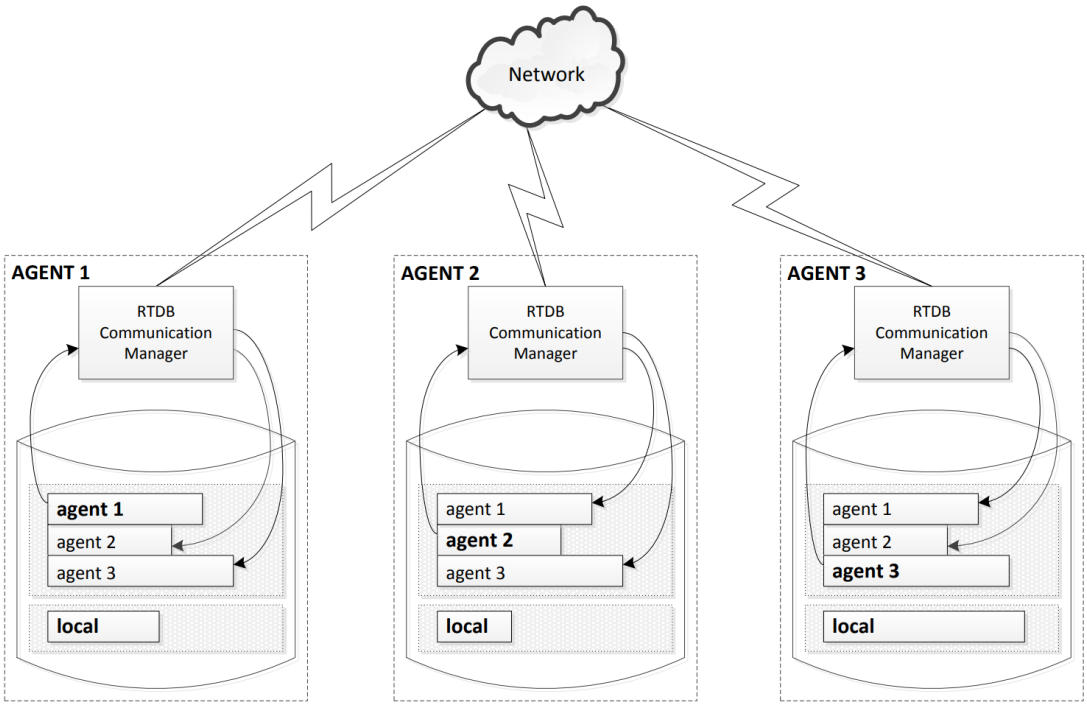


Figure 3.3: Representation of the mechanism for sharing data among RtDB instances, in this case with 3 running instances. Agent 1 communication manager retrieves data from the agent 1 shared region and sends it over the network. It also inserts the information received from the network in the agent 2 or agent 3 regions. The same strategy goes for the other agents; they retrieve information from their agent's shared region and insert data obtained from the network into the adequate shared region of the RtDB. Local area is never modified by the communication manager. This way all the shared data is replicated and local data is never sent over the network. This image was taken from [3].

It is important to mention that the system will have to consider two situations when generating the lifetime for each item. One is when the data item is produced and retrieved by processes on the same machine, and the other one is when an item is produced in one agent and is retrieved in a different one, thus in different machines. The algorithm for computing the lifetime does not rely in synchronized clocks.

On the same machine with local processes, it is simple to generate the lifetime value. The RtDB has to store the system's timestamp on the internal storage when the `put` instruction is called, as mentioned in section 3.2.1, under the attribute `timestamp`. When the instruction `get` is invoked, it has to subtract the `timestamp` stored from the actual `timestamp`, obtaining the data item's lifetime. That lifetime is returned on the instruction `get`.

The other situation occurs between different machines, meaning that the data item being retrieved was already replicated before, as shown on figure 3.4.

Taking into consideration figure 3.4, the RtDB is able to calculate the lifetime of a

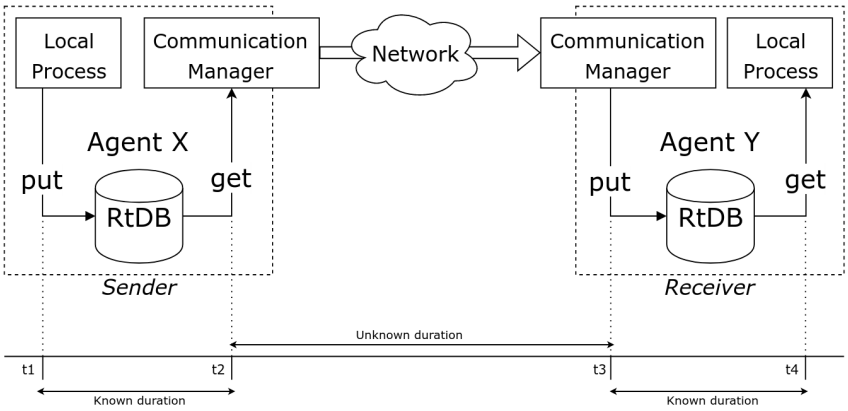


Figure 3.4: This figure represents a data item being replicated between two machines since it was first inserted by a local process in the agent X and lately retrieved by a local process in the agent Y. It also shows that the duration of the data item, its lifetime, inside agent X and agent Y is known, but when it goes over the network, it is unknown. This image was taken from [3].

---

data item while it is under the same machine, meaning that it is able to know the duration between the `put` of the local process and the `get` of the communication manager in the agent X ($t2 - t1$). It is also able to know the duration between the `put` from the communication manager and the `get` from the local process in the agent Y ($t4 - t3$). However, it does not know the duration of the item in the network (between $t2$ and $t3$) or the offset between the clocks from both computers, so the RtDB solves this problem by storing the current data item's lifetime and sending it over the network. Upon receiving it, the communication manager sums to it an estimation of the time that it normally takes to be in the network and adds to the current age.

The expression that calculates the data item's age between two machines is:

$$age = (t2 - t1) + estimation + (t4 - t3)$$

When the communication manager receives the data from the network, instead of storing the actual timestamp into the internal field of the data, it will store the actual timestamp minus the data item's lifetime until that moment. When a local process retrieves an item from a remote agent, it will have already into consideration the time that it took in the remote agent and over the network.

3.6 Summary

This chapter presented the RtDB, which is a solution based on a Blackboard architecture, meaning that all the data can easily be retrieved by any internal process. Moreover, it also allows to retrieve data that was stored by remote agents; this is done using a process, communication manager, that is responsible for sharing the desired information among a team of autonomous agents.

The RtDB architecture has many features that allows it to achieve a good performance with the necessary features for the sharing problem. As exposed in the chapter, it allows the partial replication of the storage, having data items that can be marked as shared and others as local. The internal storage of the RtDB lies completely under shared memory, and the data items are stored by copying their original memory space into the space reserved for that item in the storage. Since the items are copied into a reserved space, their size must be previously known, which comes with a limitation. It is not possible to store items that vary in space, such as dynamic structures from `C++` (e.g. `std::vector`). The user must reserve a specific amount of memory, although it might not be completely used. Another consequence is that the replications must agree on the internal storage, otherwise the RtDB might receive items that is not capable of storing.

The internal storage of the RtDB is configured through a file that is created by the programmer. After that, the programmer must run an executable to generate more configuration files that are going to be used by the RtDB. This step of generating the configuration of the internal storage is complex and non-flexible, since it is a long process and every item must have been previously declared in the configuration file in order to be replicated or used among internal processes.

This chapter also describes the method used by RtDB to estimate a data item lifetime, since the computer clocks from the instances that use the RtDB are not synchronized. The aging allows the programmer to apply decisions based on that lifetime, such as ignoring an old item. To summarize, the RtDB is a viable solution used by CAMBADA that allows to easily share information, even though it has some limitations that can be improved.

Chapter 4

Dynamic RtDB

The team CAMBADA, as mentioned before, used the RtDB for communications between the robots and internal processes. CAMBADA is a team of five soccer robots that requires communication, the same way a team of soccer players communicates with each other to understand the best strategy to achieve the objective. In this case, it might be a goal or a defense. The robots do not use voice or visual signals to communicate with each other as human players do. Instead of it, they use wireless to communicate what they are perceiving with their sensors or what they are planning to do next.

The Dynamic Real-time Database (RtDB2) is the solution that was design and implemented to solve the limitations that were mostly noticed when using the RtDB. The idea of re-implementing the middleware that was responsible for processing the data and storing it came up because it had some flaws (as mentioned in chapter 3) that could be solved with today's technologies.

As in the RtDB, the main objective of the RtDB2 is to provide a method to allow the sharing of data along several instances of a system, or even across several processes within the same system. RtDB2 should be a replacement of RtDB, keeping its functionality, while overcoming some of its limitations and doing some improvements. Thus, it should keep the blackboard architecture[4], allowing every process to store their knowledge in a centralized storage, meaning that every other process can easily access it. Since the RtDB2 must keep the functionality, it must allow the data to be shared among several agents. Therefore, the RtDB2 will use an additional process like RtDB does to share the data remotely. This process is responsible for sending the local knowledge from the blackboard to the other robots and retrieving the external knowledge that is being shared and store it.

In this specific case, the RtDB2 is going to be used mainly for the team CAMBADA. Therefore, the major changes to the RtDB correspond to limitations detected by the team in its usage. Moreover, some decisions in terms of technology used and new features to be implemented were also decided according to CAMBADA needs.

This chapter mentions several aspects that were considered during the implementation of the RtDB2. It starts by listing the existing needs that were considered important when adjusting the architecture and the technologies used. Then it explains the reason why specific technologies were selected instead of others that were similar but got excluded.

Moreover, it expounds the software structure used to achieve the requirements and some details about the implementation, such as including a feature to allow the sharing of data items with a given periodicity and phase. This chapter also covers how the RtDB2 grants a backward compatibility with RtDB.

The RtDB2 works in a similar way as RtDB did. It uses a blackboard strategy [4] just like RtDB did (as mentioned in 3.2). Every process uses a shared data region where it is allowed to read or write data on it to solve a common problem. In this case, every process will use the RtDB2's API in order to write items to or read items from the data storage, which has gotten an update when compared with the RtDB's API (as shown in section 4.5 where the RtDB2's architecture is discussed). Similar to the RtDB, the newAPI will also have methods to insert or retrieve a data item from a specific identifier, but it also has methods that allow to collect or store batches of data in a single operation.

4.1 Requirements

The RtDB2 has many requirements that need to be fulfilled to be able to replace the RtDB, some of them easier to achieve than others. Some of the requirements considered here were not necessarily features that RtDB already had, but new required features.

In order to have a clear and distinct list of requirements (or even characteristics that help to define the requirements), they will be grouped into two lists. One contains requirements that must exist to keep the behavior from RtDB and another one accomplishing the possible improvements to the system. The requirements needed to keep the behavior of the RtDB are:

- Items that are shared among other instances of the RtDB2 and items that are simply local, which are never sent over the network, although they can be retrieved by other processes with direct access to the RtDB2. This was used by RtDB and must be kept in the system.
- Data item must have a known lifetime associated with it. It must be known how long an item has been created or received a new value in order to be able to discard outdated values.
- The impact caused by the methods used to retrieve and insert data into the storage must be negligible when comparing with the time taken by the robots logic. The RtDB uses shared memory to achieve a good performance, so the operations to insert and retrieve have almost no impact in the time consumed by the agent. Therefore, it is important to guarantee that the time consumed by the RtDB2 operations is minimal when compared with the deadlines of the agent's logic.
- The new API[1] must include the old one on the beginning, so the programmer is not affected by the RtDB2 changes. This means that the RtDB2 must grant full

[1]New API is always referred to the API from the RtDB2 and old API is always used to mention the one from the RtDB

backward compatibility with the RtDB, so that all the code that has been developed until now does not require any changes and the old API is still available. This is a temporary feature that might be deprecated later after the RtDB API is no longer used.

The requirements for improvements and new features are:

- The storage should allow to store dynamic data structures, such as `std::vector` from `C++`. As mentioned before, the RtDB does not allow the storage of this type of items.
- Data compression should be possible. Including the possibility to compress the data might be useful in some situations, such as sending data over the network. This is useful in CAMBADA, since the amount of data that can be sent through the network is limited during a match, imposed by the league rules.
- Share data between different programming languages instead of being restricted to a single language. The RtDB is restricted to a single language because it is a simple shared-memory middleware that stores every data structure directly from memory using `memcpy`. Therefore, every instance of the RtDB must have the exactly same description. Otherwise, it will fail to load the item.
- The load of the RtDB2 must be tolerant to small drifts in the data structure definition among several instances, such as a missing field in a structure or a renamed field in a structure received from another agent. In the RtDB, small changes in a data structure would cause the storage to fail its initialization; the RtDB tries to copy the item into the allocated shared memory space, although it will crash when the structure size differs from the allocated space.

As an important note for the storage, it is not required to save the data storage between executions, so the data can be discarded when the system is turned off. Therefore, the storage can be volatile. However, there are benefits of using a storage that keeps the data between crashes or shutdowns, such as being able to remember the last state without having to receive any data.

4.2 Strategy used by RtDB2

In RtDB, the method used to share the data structures locally or remotely has no differences. It means that both situations just copy memory blocks from a place to another one. In other words, when receiving data from the network or storing data, the RtDB picks the block of memory where the data structure was initially created and copies it into the internal storage, using `memcpy`. Similarly, when sending data over the network or retrieving data, the RtDB picks the block of memory allocated for that structure and copies it to a destination pointed by the function's caller. However, copying memory blocks from a place to another might not be the best strategy.

The fact that it copies the block of memory where it is stored and not every language represents a data structure in the same way will not allow it to be compatible across several languages. Another problem that comes with it is that every agent that runs the RtDB must have the same definition of that structure. This issue means that a data structure can not have one less or additional field on it, otherwise it will fail to load since the source has less bytes than the destination. Also, it might not succeed to get the correct values when the system architecture differs, such as having a little-endian representation in one machine and big-endian in another one. Even if it is possible to extract the data in a given situation, it is not the desired method to store the data, since it is not flexible.

In figure 4.1, it is possible to verify the existence of two different situations: storing data; sending data over the network. In the RtDB2, all the data that is desired to be stored must be serialized. That step is done in a transparent way to the programmer. It means that the programmer simply invokes a `Put` function as before (with the same signature as the previous one from the RtDB) and its implementation will serialize the data. When the programmer decides to obtain data through a `Get`, the data will be deserialized and simply returned.

When sending the data over the network, there is an additional step: compression. Even if it is not a required step, in MSL the amount of data that is sent over the network is metered and limited by the rules, so it is important to minimize the used bandwidth as much as possible.

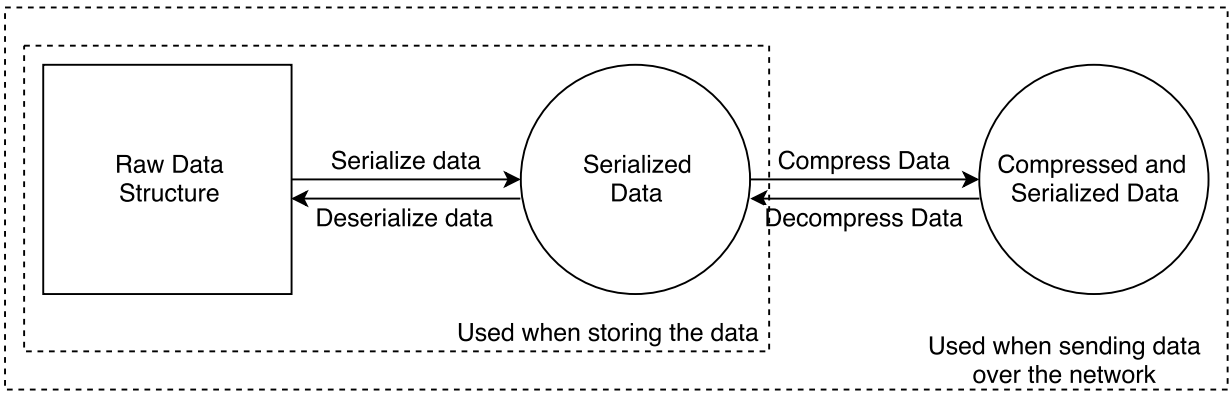


Figure 4.1: RtDB2's strategy to store data and send it over the network. The RtDB2 serializes the data to be able to store it, and when the data is retrieved, it gets deserialized. If the data is required to be sent over the network, the RtDB2 is going to compress the already stored data (that was previously serialized) and return it.

In the CAMBADA, there is a process known as communication manager process (`comm`) that is going to pick the local copy of that agent's shared data from the RtDB2, compress it and send it as a batch through the network. The communication manager from the other agents will simply be waiting for incoming batches. After receiving a batch of data, the communication manager will decompress it and store it in their internal RtDB2 storage. Compression only occurs when sending data over the network, and it does not involve any serialization or deserialization because it simply picks what is already stored, that is

already serialized, compresses and sends it. Therefore, on the receiver's side, it is only required to decompress since the data is already serialized.

Cross-language compatibility of RtDB2

The fact that all the data stored is being serialized allows it to be deserialized in other programming languages, provided that the programming language is supported by the serializer API. There are two ways to catch and use the data from the RtDB2 in another programming language:

1. When it is being sent over the network. It will be required to decompress the data received and then deserialize to obtain the original raw data structure. Moreover, the compressor and the serializer must be supported by the desired programming languages to be able to reverse the process in another language. This allows to bring some tools related with network, such as a network analyzer implemented in any other language, as long as the compressor and the serializer support it.

2. When the data is stored in one of the RtDB2 instances. The storage used must be loaded in the other language and then deserialize the desired raw data structure, or insert a serialized item. Furthermore, the storage and the serializer must be supported by the desired programming languages. The storage is required to read what is written in it and the serializer to deserialize the data structures. This method allows the creation of tools like a content inspector to find what is the value of a given field in the storage.

It is important to state the both methods are catching the data in completely different situations, one is listening to the data that is being shared and the other one looks directly into the storage. The one listening will be obtaining only the shared data items. The other method can get the local and shared data items from the specific agent where it is running and the shared data from the other agents.

4.3 Critical flows in a CAMBADA's agent

Since CAMBADA is the main user of the RtDB2, it is important to understand some of the scenarios of how the information is shared among all the agents. Or even scenarios that help to understand how the data structures are going to be shared with the RtDB2 and how it was with RtDB.

4.3.1 Cycle of a CAMBADA's robot and its processes

Internal processes

In CAMBADA, each robot runs several internal processes as shown in figure 3.1 (from RtDB chapter), such as:

1. `hwcomm` is a process that is responsible for communicating with the low-level layer through the USB/CAN gateway.

2. `agent`[2] is the process responsible for decision, coordination and reasoning. So it is the process that will be handling decisions based on the input received. In this case, it will be taking decisions from the content inside the RtDB2 since it is where most of the processes store the information that they perceive from the sensors. The RtDB2 does not contain only sensorial data; it contains already processed data and not raw values taken directly from the sensors.

3. `vision` is responsible for obtaining the frames from the digital camera, process them and store the relevant information that was perceived into the storage.

4. `comm`, also known as communication manager, is the process responsible for handling the data from the network and sharing it to the network as well. It is composed of two threads; one thread that is responsible for obtaining the data that is being sent by others robots and insert it inside the local RtDB2; and the other that will grab all the shared fields from the local RtDB2 and send it asynchronously through the network using multicast.

A robot has more internal processes than the ones mentioned before, although they do not have relevant impact in the cycle. One of those processes is the `monitor` that is responsible for keeping track of all the processes from a robot and relaunch them if something goes wrong with any of them.

Cycle of a robot from CAMBADA

A robot has many processes running asynchronously, but it also has some of them running synchronously because they might need to wait for some data that is perceived or obtained by another process. An example of a synchronous case is the main cycle of an robot from the CAMBADA, as shown in figure 4.2.

This cycle has clearly three processes that depend from each other: `Vision`, `Agent` and `HWcomm`. The cycle in this situation is well-defined: the `Vision` obtains a frame from the camera, processes it and then stores the processed data into the RtDB2. Only then the `Agent` starts deciding what is the next step to take, since it requires the information extracted from the image. Once the agent has decided what it wants to do next, the data will be stored into the RtDB2 and obtained later by `HWcomm` on the start of the next cycle. It is only processed later on the next cycle, so that only `Vision` and `Agent` have impact to the cycle's duration and not `HWcomm` as well. This will reduce the temporal jitter when communicating with the low-level layer [34]).

[2]This process, known as '`agent`' should not be confused with the entity agent, that is normally used to refer to the whole system used by a robot. Whenever the process '`agent`' is mentioned, it is attached with the word process to avoid ambiguities.

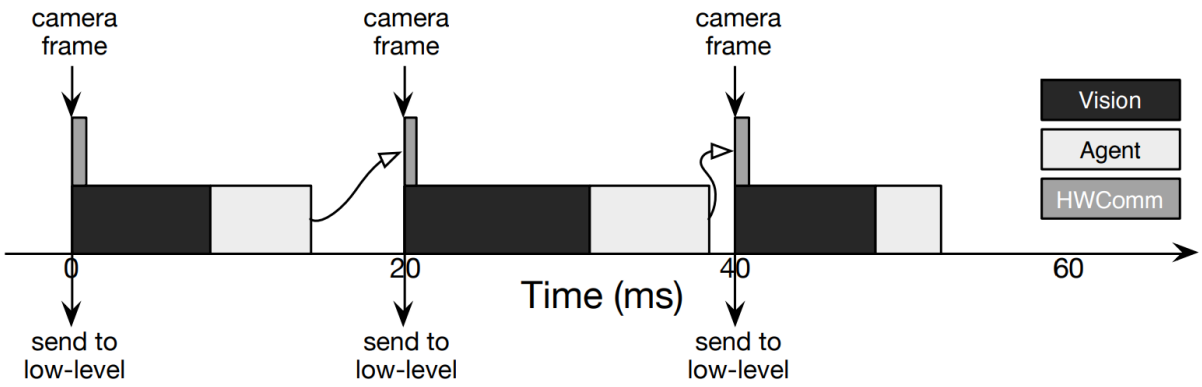


Figure 4.2: Representation of a CAMBADA's agent cycle. It starts when a new camera frame is received and then two processes run simultaneous, `HWcomm` and `Vision`. After those two processes have ended, then `Agent` process starts. At the end of the cycle, defined at twenty milliseconds, `hwcomm` will just send the information processed to the low-level layer. Image taken from CAMBADA team description paper [34].

As it is possible to verify, there is a lot of communication between the processes by using the storage from the RtDB2. Therefore, it is important that the method to store the data into the RtDB2 and obtaining it is much faster than the cycle's deadline, which is twenty milliseconds. It is important that the RtDB2 is not what mainly consumes the cycle, leaving enough time for the rest of the logic.

As mentioned before, there is an additional process that must be considered into this flow, the communication manager, that is responsible for sending all the shared data from the RtDB2 to the other agents. In section 4.2, it is possible to verify that now there is an additional step, compression, that is applied when sending or receiving data through the network making the operation slightly more expensive.

4.3.2 Sharing data between two agents

The RtDB2 has considered that sharing an item locally or remotely has different requirements in terms of data size. When sharing a data structure locally among the internal processes, there is less concern about the space that a data structure is going to occupy, but when dealing with a limited bandwidth network, there is a great concern about the amount of data that is sent through the network.

In image 4.3, it is possible to verify a flow between two agents, where agent 1 has inserted a shared item into its storage and agent 2, that is not in the same machine, retrieves it later.

Any interaction between two or more remote processes in different RtDB2 instances requires at least five steps, as shown on figure 4.3, to be able to share the data structures:

1. Initially, the process that wants to share its information stores it using a `Put` method where the data will be serialized and inserted into the storage.

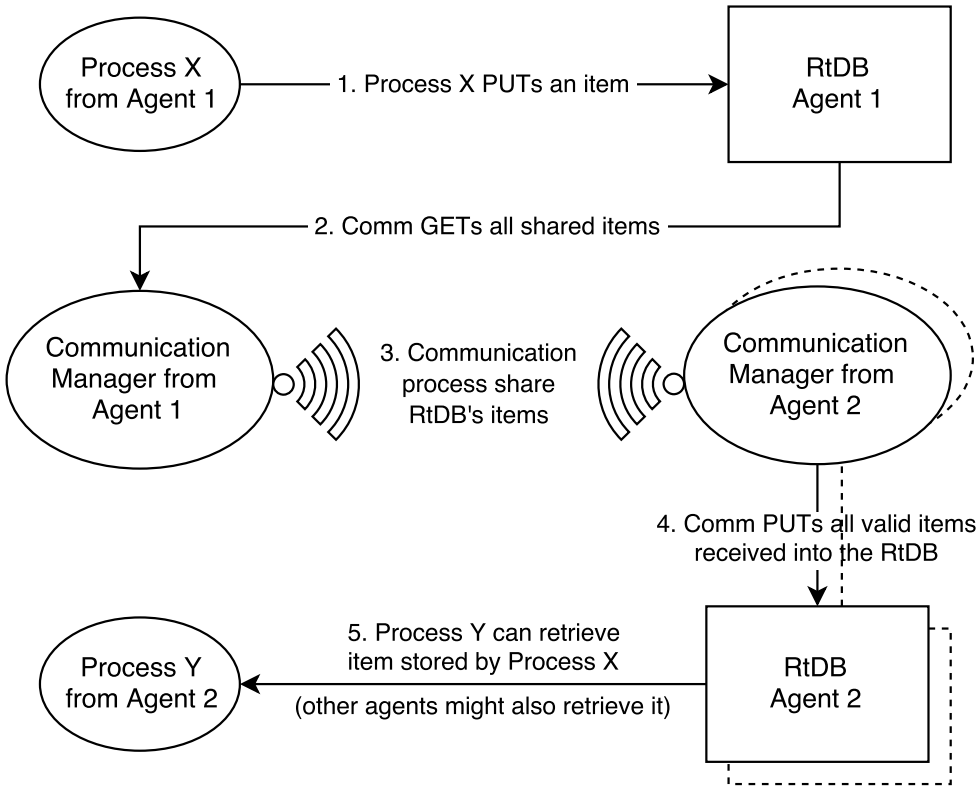


Figure 4.3: Representation of a process X indirectly sharing a data item with a process Y. The process X simply inserts the item in RtDB2 and the communication manager obtains those items and send it over the network, so that the other communicating manager in the destination's agent can store it in the second agent's storage. After that, process Y can retrieve the item that was stored by process X.

---

2. Communication manager will obtain all the shared items from the local storage in order to send it through the network. Differently from the RtDB, it will obtain all the data from the storage in a single operation and then compresses it, meaning that the `Get` operation is different from the one used by a normal process. This one compresses the data that was already serialized, while the normal `Get` deserializes the data to obtain the original data structure. Therefore, it is completely different, this operation is known as `GetBatch` instead of `Get` and it obtains all the data that requires to be shared from the storage.

3. Communication manager that has the information sends all the data to the multicast channel for anyone who is listening. Even though it might not be the best method to grant that the information is successfully shared, it is not that important, since it will most likely share newer information at every agent's cycle, twenty milliseconds;

4. The communication manager that has obtained the information will decompress the data and use `PutBatch` to insert it in the storage as a single operation ready to be

obtained by any process in the remote agent.

5. At last, a process retrieves that information using `Get` operation. The operation is responsible for deserializing the data structure and return it.

Ideally, there will be much more insertions and retrievals from the storage by normal processes that require the information than from the communication manager. The fact that the data is not compressed every time it is inserted into the storage, but only when shared by the communication manager, makes it easier to understand why the compressor does not require to have more performance in terms of response time than the serializer. Moreover, the communication manager will only compress the shared items once as a huge chunk of data, and not for every single item that requires to be shared.

The serializer will be required to serialize or deserialize the data every time that a new data structure is inserted or retrieved, respectively, by a process. Considering that the serializer will affect most flows that require to share information (locally or remotely), it must have a very low response time. As an example, it must be a barely noticed operation during the cycle from CAMBADA's agent described in section 4.3.1.

4.4 Technology Selection

In this system, it is important to select the technology that is going to be used before implementing anything, since there is some technology that affects the way that the implementation is done. As referred in section 4.2, there are two steps that might occur: serialization and compression. These two steps will be done by externals tools; in this case, a serializer and a compressor. There is an additional tool that it is required and it is not so explicit, the storage. It could be done just like the RtDB and use shared memory, storing a structure directly into it, but the idea is to obtain some improvements with these changes. Consequently, it will be required to pick the three different tools that fit better for this situation: storage, serializer and compressor.

4.4.1 Storage

When comparing this system with its older version, the RtDB2 will use a database to store its data instead of using shared memory. This has some advantages, such as not having to deal with the location where data is going to be stored in memory and its organization, or concerning about concurrency access. Also, when using a database, it will have specific optimization to achieve good performance, which is a great benefit when dealing with a system in a real-time environment.

The storage is one of the important tools that must be picked carefully. There is a huge variety of databases that can be used to store the data, although the chosen one must be picked according to the scenario where it is going to be used. It is impossible to state that there is a perfect solution; every system has its perks and must be adapted with the requirements of the system.

Considering the system requirements, some assumptions were made from the beginning. NoSQL was the only database type that was considered, since there is no need to define relations between data. Another assumption was to only use a NoSQL database that can work as a key-value store; there is no necessity to use a more complex system than required, it might bring overhead (as mentioned in section 2.3).

The decision that was applied to key-value store databases was also applied to distributed databases. The RtDB already includes a system that replicates the data; it is done through the communication manager. This replication also includes specific mechanisms to generate the lifetime of a specific item, which allows to know if a given item is usable or if it is outdated. Moreover, distributed databases were excluded due to the current replication method, meaning that communication manager and lifetime are still used in a similar way as before with minor modifications.

Even when selecting only NoSQL key-values store, there is an extensive list of possibilities. Consequently, some requirements were defined in order to be able to exclude some of them:

1. The library must be completely free with a non-copyleft license. A copyleft license offers the right to copy and modify the source code freely (depending on the license), although it requires that all derivative source code from it must have the license rights preserved. It means that if the license was free to modify or copy, it must keep that agreement in derivative works. In the case of a non-copyleft, there is no requirement to keep the same license rights in modified or extended works. It is also known as a permissive license.

2. The storage library must be a maintained. It can not be outdated, thus it should keep be receiving regular updates.

3. The key-value store should be of in-memory type. An in-memory solution typically achieves better performance with the downside that it may not guarantee that the data is non-volatile, as mentioned in section 2.2. However, the RtDB2 does not require to save its state between executions.

4. It must work in a multi-process and multi-thread environment. The CAMBADA only uses different processes to interact with the RtDB, although it is a great benefit to have the possibility to be able to interact with the storage in a multi-thread environment as well.

5. It must support ACID transactions. It is important that the storage has a concept of transactions, so it allows to create retrieve batches of data without any other process interfering with that data while it occurs. It grants that each item in the batch belongs to the same situation, being useful for the communication manager when replicating all the data. It might also be useful in other situations, such as storing only data if all the previous data was stored successfully.

Table 4.1: Comparison between the key-value store candidates for RtDB2. The candidates are compared according with the following requirements: license; maintenance; ability to store in-memory; possibility to work in multi-process and multi-thread environment; transactions with ACID properties and a `C++` API.

	License	**Maintained**	**In-memory**	**Multi-process**	**Multi-thread**	**ACID**	**C++**
LMDB	OpenLDAP 2.8	Yes	Yes	Yes	Yes	Yes	Yes
LevelDB	3-clause BSD	Yes	No	No	Yes	No	Yes
Redis	3-clause BSD	Yes	Yes	Yes (1)	Yes (1)	No	Yes
RocksDB	3-clause BSD	Yes	Yes	Yes (2)	Yes	No	Yes
BerkeleyDB	GNU AGPL v3.0	Yes	Yes	Yes	Yes	Yes	Yes
UnQLite	2-clause BSD	Yes	Yes (3)	Yes (2)	Yes	No (3)	Yes
BangDB	3-clause BSD	Yes	Yes	No (4)	Yes	Yes	Yes
Kyoto Cabinet	GNU GPLv3	Yes	Yes	No (5)	Yes	Yes	Yes
c-treeACE	Commercial	Yes	———	———	———	———	———

(1) Uses an additional process known as broker; (2) Only allows a single process with write mode; (3) Does not grant ACID properties in in-memory mode; (4) Only when in-memory mode is disabled; (5) Does not allow writers or readers to connect to the storage while a writer is connected;

---

6. It must have a direct API implemented in `C++` or at least an API with bindings for `C++`.

Table 4.1 contains a comparison between some candidates to the storage of the RtDB2, where they are compared based on the requirements previously mentioned.

It is possible to verify from table 4.1 that there is some variation in the features supported by each database. However, there are some store databases that can easily be excluded, such as `c-treeACE`, which is a commercial product and does not have a free version. Some of the databases support all requirements, although they also have some limitations when using all the features together. `BangDB` is an example of that case: it does support multi-process, although it becomes unusable when using in-memory storage.

`RocksDB` and `UnQLite` support multi-process with the limitation that only one of those processes may be used for writing, which is not acceptable in the case of the RtDB2. `Kyoto Cabinet` is similar to this situation, but it does not support multi-process when there is a process that requires writing; all of the other processes can not even be connected to the database. `Redis` uses an additional process, known as broker, that is responsible for sharing the data, locally and remotely; this is not exactly a limitation, however there is a preference for not having an additional process when sharing data locally. It also supports multi-process and multi-thread. These mentions for each of the databases are limitations related with requirements for the RtDB2.

`LMDB` and Berkeley DB (`BDB`) are two databases that support every one of the requirements with the exception of the `BDB`'s license that is copyleft, meaning that the derivative works of it must be open-source as well. Consequently, both are good candidates for the

Table 4.2: Benchmark comparison between LMDB and BDB with basic operations of read and write. All the values mentioned in this table are expressed in microseconds that a single operation has taken. Moreover, the database with the smaller values has performed better in that situation. Several tasks were performed during this benchmark: sequential write (fillseq); sequential write of batches (fillseqbatch); random order write (fillrandom); random order write of batches (fillrandbatch); overwrite the existing values (overwrite); random read the values in the database (readrandom); read the values sequentially (readseq); read sequential but in reverse order (readreverse). Each of these tests has average, standard deviation, median and maximum associated to it. Each test was written or read one million key-values, each key had sixteen bytes and each value had one hundred bytes. The batches were composed of one thousand entries.

	BerkeleyDB				**LMDB**			
microseconds/op	**Average**	**Standard Deviation**	**Median**	**Maximum**	**Average**	**Standard Deviation**	**Median**	**Maximum**
fillseq	13.20	995.03	7.70	579289.91	1.69	1.20	1.35	52.21
fillseqbatch	6.87	308.29	3.32	65860.98	0.41	1.05	0.55	49.83
fillrandom	48.19	779.96	15.39	515967.13	2.95	1.61	2.85	66.04
fillrandbatch	50.08	912.55	11.41	541944.02	2.07	3.72	1.66	169.04
overwrite	50.05	1463.06	15.41	973196.98	5.99	0.93	5.67	37.19
readrandom	6.29	6.43	5.78	4297.97	1.12	0.39	0.71	30.99
readseq	0.98	0.79	0.63	174.99	0.09	0.31	0.51	26.94
readreverse	0.97	0.76	0.63	87.98	0.09	0.29	0.51	31.95

storage in the RtDB with a higher preference for the LMDB. In order to find out which one should be used, some benchmarks were done.

LMDB and BDB performance

Since the databases that have the required features are LMDB and BDB, some tests were performed in order to find out which one would perform better. The benchmark tests performed were based on original tests developed by Google[3] and also adapted by LMDB[4]. The benchmark source code was executed in the typical hardware used by CAMBADA (which is later described in the results, section 5.1). In table 4.2, it is possible to compare the results obtained between LMDB and BDB.

In the table 4.2, the database that achieves smaller values for a given test has performed better in that situation. In this case, it is clearly visible that the LMDB outperforms the BDB in every operation. The maximum values expressed by the BDB during write operations are extremely high. For example, in the sequential write (fillseq), the maximum is 579 milliseconds per write, meaning that one of the writes took half a second to complete. However, there might be several factors that might affect a maximum value, such as having many background processes owned by the system.

[3]LevelDB benchmark developed by Google: https://github.com/google/leveldb/tree/master/doc/bench

[4]LMDB benchmark official results from the adapted LevelDB benchmark: http://www.lmdb.tech/bench/microbench/benchmark.html

When comparing the standard deviation with the average of a specific test, it is possible to verify that the variation of data is much higher on the BDB tests than on the LMDB. It means that the LMDB performance is more likely to be closer to the average than the BDB, thus more stable response times.

Therefore, the LMDB was selected as the storage for the RtDB2 due to the performance results when compared with the BDB, and the positive factor of having a non-copyleft license.

4.4.2 Serialization

Serializer is one important tool, almost as important as the storage. The data items are going to be serialized whenever they are introduced into the database, and deserialized when they are retrieved from the database, as explained in section 4.2.

Serialization is the act of translating a data object or a data structure into a well-known format. Deserialization is the reverse process to obtain the original object. As previously mentioned in section 2.4, there are many features that must be considered when choosing the right serializer, and these mentioned features are the ones that are going to be used to create a comparison between the serializers. This comparison includes the following features: possibility to serialize in one programming language and deserialize in a different one; schema evolution of the serializer; possibility to serialize the data into binary or human-readable encoding, or even both; and the necessity to define a schema to be able to serialize. In table 4.3, it is possible to see a comparison between the most important features of several serializers that are available in C++.

By considering the Cross Language Compability (CLC) as a necessary feature, it is simple to exclude three serializers: Capnproto, Boost and Cereal. One major feature from the RtDB2 is the ability to work in several programming languages, thus it is important to have CLC. Another important feature that a serializer must have is to be able to serialize in binary form, which all of them provide. Moreover, human-readable (or text enconding) is only useful for debugging purposes, so it is not a high priority feature to have.

Besides encoding and CLC, there is also the possibility of discarding the usage of a schema. Not using a schema typically allows the developer to approach the problem more directly, without having to think about generating a schema for a recently created data structure. However, there are always impacts of not having a schema. In the case of Binary JSON (BSON), it does not have any direct impact, since it is a JSON, but with binary encoding; this means that it does not apply any optimization to the data, thus the data will still have a larger size when comparing with other serializers. In the case of MsgPack, it allows to skip the usage of a schema, although it will require the usage of an intrusive macro. That macro is required in each structure to show MsgPack which fields are going to be serialized in a given structure, and how they should be serialized.

At last, schema evolution is the designation used to describe how strict is a deserialization when the data structure has slightly changed after the serialization. In this case, several tests were made to verify the schema evolution of each serializer: renaming field, adding a new field, removing a field and type inheritance. The only serializer totally

Table 4.3: Comparison between the serializers candidates for the RtDB2. The candidates are compared according with the following features: Cross Language Compability (CLC); freedom in terms of schema evolution; serialize into binary form; ability to encode in human-readable text; and necessity of using a schema or not.

	CLC (1)	**Schema Evolution**				**Binary encoding**	**Text encoding**	**Requires Schema**
		Renamed Field	**Field added**	**Field removed**	**Type Inheritance**			
Thrift	Yes	Yes	Yes	Yes (2)	No	Yes	Yes	Yes
Protobuf	Yes	Yes	Yes	Yes	Partial (3)	Yes	Yes	Yes
Avro	Yes	Yes	Yes	No (4)	Partial (3)	Yes	No	Yes
Msgpack	Yes	Yes	Yes	Yes	Yes	Yes	No	No
BSON	Yes	No	Yes	Yes	No	Yes	No	No
Flatbuffers	Yes	Yes	Yes	No (4)	No	Yes	Yes	Yes
Cap'n Proto	No	Yes	Yes	No (4)	Partial (3)	Yes	No	Yes
Boost	No	Yes	No	No (4)	No	Yes	No	No
Cereal	No	Yes	No	No (4)	No	Yes	Yes	No

(1) Cross Language Compability (CLC) - The serializer is able to serialize in one programming language and deserialize in another one; (2) A field might be removed if it was previously marked in the schema as optional; (3) A type might be converted from float to integer, but not from double to integer (due to the number of bytes used); (4) Works only for the last field in the data structure;

permissive in this aspect is `MsgPack`, which allows any kind of modification to the data structure by mapping each field when serializing, thus occupying more space than what would normally be without mapping each field.

`ProtoBuf`, `Thrift`, `BSON` and `MsgPack` are good candidates to the RtDB2 serializer, considering that all of them meet almost all the required features. In terms of schema evolution, they are partially permissive by only failing when changing the data type of a given field, with the exception of `MsgPack` that handles the problem correctly. Another great advantage of `MsgPack` is that it does not require a schema to serialize or deserialize the data.

The `MsgPack` handles the schema evolution specially: it allows the definition of a macro that maps every field with its value during the serialization, meaning that the field name is contained somewhere in the bytes of the serialization result. It has the benefit of allowing severe modifications to the structure and still be able to deserialize the object. However, it has a great impact in terms of the size of the serialized data, since it will contain the field name. On the other hand, there is also a macro that has the possibility to not map the field name into the serialized result, although the schema evolution will not be so flexible.

Due to this flexibility, the `MsgPack` is the serializer selected for the RtDB2. The strategy used is a hybrid solution between the macro that maps the field name and the one that does not map. In other words, the structures that are not likely to be changed are going to be serialized without the field name, such as two dimensional points that contain a field

named “x” and “y”. On the other hand, the structures that are more likely to be changed are going to map the field name, allowing to benefit from the flexibility given by `MsgPack`.

4.4.3 Compression

The RtDB2 is going to use a compressor to send as few bytes as possible over the network. In this case, several serialized data structures are going to be sent after getting compressed. The compression is going to be applied to sensible data that can not lose or change any of its bytes, otherwise it corrupts the result of the serialization; this means that a lossless compressor is required.

It is quite important to know which information will be compressed in order to pick the best compressor type and tool. Since the only information that is known about the data is that it was serialized before, it is difficult to make any assumptions about what type of compressor should be used. The only known fact is that the serialized data is binary and might vary a lot. Considering that the `MsgPack` was the picked serializer, as mentioned in the previous section, it is also important to mention that it is going to map several fields, which means that these fields are going to appear in the bytes with their name. Moreover, several structures are being sent during a single compression. Therefore, there is a strong possibility that a field name to appears repeated during the compression due to the mapping done by `MsgPack`.

Consequently, it is going to be selected a compressor with a Lempel-Ziv algorithm [31] or derivative, since it is used for general purpose and it is dictionary-based, which may help to reduce the repeated field names generated by `MsgPack`. The selection of the Lempel-Ziv (LZ) compressor was done using an open-source tool known as `lzbench`[5] by running several tests under the system used on CAMBADA, in order to find the best solution.

This tool, `lzbench`, allows to run several compressors that implement a Lempel-Ziv algorithm or derivative over a specific file to find their performance. In this case, the file used for the tests was a batch of serialized data structures used by CAMBADA, to find the best tool that is fast enough and also compresses efficiently. Table 4.4 shows a comparison among several compressors dictionary-based of the compression result in terms of compression speed, decompression speed and compression ratio. On a side note, it is important to mention that the term compression ratio used is expressed as follows:

$$Compression\ Ratio\ (\%) = (Compressed\ Size\ /\ Uncompressed\ Size) \times 100$$

As an example of the compression ratio expression, considering that the original data has 1000 bytes and gets compressed to 200 bytes. The compression ratio is (200 / 1000) $\times$ 100 = 20%, one fifth of the original data.

The table 4.4 is a reduced version of the one mentioned under the appendix B.1, since the benchmark includes many tools that could have been used. The reduced version does not include variations of the same tool, showing only the best of variations for that specific tool. Many compression tools were also excluded because they did not have the best

[5]lzbench GitHub Page: `https://github.com/inikep/lzbench`

Table 4.4: Comparison between the dictionary-based compressors candidates for the RtDB2 using `lzbench`. The candidates are compared according to their compression ratio and their compression and decompression rate. The tests were made under a CAMBADA's system with a batch of serialized data (randomly picked), that is the typical data that is going to be received by the compressor in the RtDB2. The values from the table under the columns "Compression" and "Decompression" are expressed in Megabytes per second. The values under size are expressed in bytes and the ratio is expressed in percentage.

	Compression (MB/s)			Decompression (MB/s)			Size (bytes)		
Compressor name	**Maximum**	**Average**	**Median**	**Maximum**	**Average**	**Median**	**Original**	**Compress**	**Ratio**
blosclz 2015-11-10 -6	358.26	174.62	193.78	1647.64	269.85	214.4	2338	1342	57.4
brotli 2017-03-10 -0	122.74	120.6	120.71	178.02	175	174.71	2338	1339	57.27
fastlz 0.1 -1	252.35	224.29	226.35	1115.99	205.54	202.06	2338	1342	57.4
lizard 1.0 -22	186.1	182.91	183.89	2603.56	223.82	215.6	2338	1319	56.42
lz4 1.7.5	569.55	169.84	168.57	3238.23	310.08	220.69	2338	1326	56.72
lzf 3.6 -0	153.98	149.74	153.66	955.85	201.24	190.51	2338	1338	57.23
lzjb 2010	358.92	197.02	186.94	709.13	223.56	180.19	2338	1369	58.55
lzo1c 2.09 -9	128.61	123.83	125.76	1715.33	234.9	218.61	2338	1322	56.54
lzrw 15-Jul-1991 -5	109.45	107.41	107.87	294.64	186.76	204.51	2338	1372	58.68
slz_zlib 1.0.0 -2	198.07	194.32	195.26	250.37	191.4	217.45	2338	1291	55.22
yappy 2014-03-22 -10	123.9	121.26	121.61	4182.47	361.81	228.79	2338	1309	55.99
zstd 1.3.1 -2	117.72	115.89	117.44	289.57	160.03	208.69	2338	1165	49.83
shrinker 0.1	496.71	129.32	158.4	2461.05	320.32	218.4	2338	1305	55.82

speeds and compression ratio. Consequently, the reduced list includes only compression tools with average compression speed of at least 100 MB/s, a decompression speed of 150 MB/s and a compression ratio lower than 60%. The idea was to filter as much as possible the compressors by keeping the most relevant.

Considering table 4.4, the `zstd` library stands out by having a compression ratio under 50%, that is at least more 5% than the others mentioned in the table. The `zstd` compression and decompression speeds are not the best ones when compared with the others in the table, although it does not mean that the speed is not enough for the system requirements. In terms of speed, there are many good results such as `fastlz` with an average of 224.29 MB/s when compression or `yappy` with a decompression speed of 361 MB/s.

The `zstd` has also one advantage that was not tested in here: the speeds and the ratio can be improved even more. The library allows to use a pre-trained dictionary, meaning that it is possible to train the compressor with data that is going to be typically received by it. However, that dictionary will have to be shared among all the replications, otherwise the compression results will be different from each other and unable to be decompressed by each other. Consequently, the RtDB2 is going to use `zstd` as default.

4.5 Architecture

This section includes some explanations about the architecture decisions. It also discusses how the replication process is done; how the lifetime of a given item is generated locally and remotely; what is the periodicity and phase of an item; and how the configuration file works in the RtDB2.

4.5.1 Shared and local instances replication

As a legacy and required feature, every data item in the RtDB2 does have a characteristic that defines an item as shared or local. As mentioned before, local is an item that will not be sent to other instances of the system, it is only shared among local processes. Any item that does require to be shared with a remote instance must be declared as a shared item. The RtDB2 provides instant access to an item that has been shared by a remote instance, meaning that the last shared value in a given key is always stored locally. As previously mentioned, the communication manager will be responsible for listening and storing items that are being sent by remote instances.

Assuming that there are several instances of the RtDB2 running, there might be the same key represented in more than one instance, such as `ROBOT_WS`. Each instance has an unique identifier to differentiate them, it is explicitly chosen when creating the RtDB2 through the constructor (as mentioned later in the API from the RtDB2, at section 4.6.1). Therefore, each RtDB2 instance has an unique identifier. The RtDB2 must have a way to be able to identify the instance that originated a value for a given key, thus each key must also be identified with the storage instance. This problem of having repeated keys can be solved in two distinct ways:

1. Using a single database storage where the key must have the RtDB2 instance identifier alongside with the key. An example of a key can be `4_ROBOT_WS`, where the first value before the underscore identifies the instance and the second one identifies the key being inserted. However, this solution does require parsing every time that the communication manager wants to retrieve keys inserted by the local instance in order to send them. It will have to run through all the items in the database and find out the ones that have the identifier of its own instance.

2. Using several database instances where one of these is a local instance (used by local processes, it can include items that are going to be shared and local items) and the rest of the database instances, known as remotes, are used for the shared data. These remote instances are used to store the data items received by the communication manager, one for each remote agent.

The second option was the chosen one, since it does not involve any parsing from the communication manager. It will only have to open the database from its RtDB2 identifier (local instance), find the shared items and send them over the network, although this solution has a flaw. The communication manager can not tell directly which items are shared or not without having to iterate through them.

Therefore, the database containing the local data was separated in two: a database for the local items that are never shared with remote instances, and another one for items that are shared with other instances. This way, the communication manager does not have to iterate through the database: it will simply have to pick all the items in the database and send them, as shown in figure 4.4.

Figure 4.4 shows two gray boxes in the RtDB2 storage that correspond to the databases that are modified by the internal processes of the agent, excluding the communication

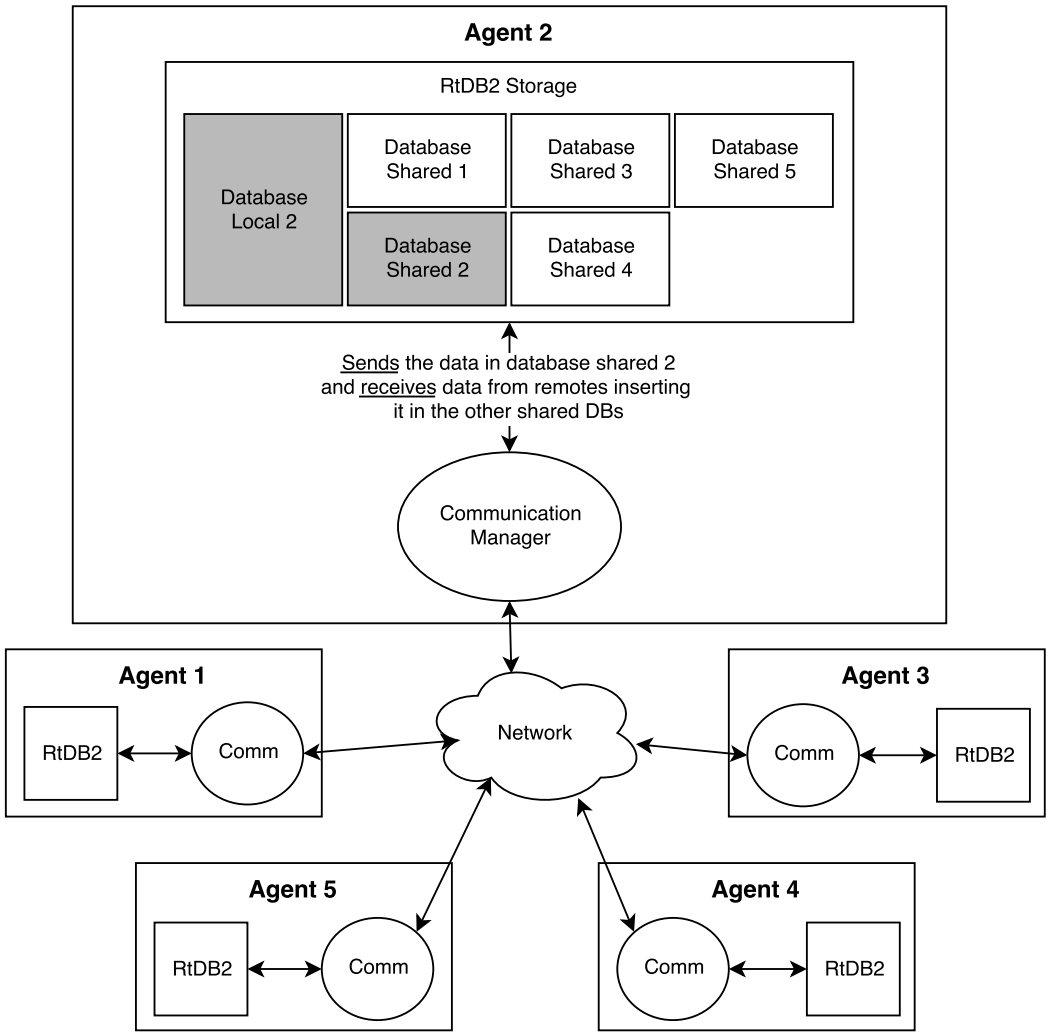


Figure 4.4: Representation of how the RtDB2 replicates its data items along the other instances. This image shows how the storage handles replication of its and remotes data by using the communication manager. The gray boxes represent the databases that are filled by internal processes that contain the agent's logic. The other boxes inside the RtDB2 correspond to the shared items from the remote agents that are filled up by the communication manager when it receives the data.

manager. The database local 2 is never accessed by the communication manager, since it only uses the shared databases. The communication manager has two different behaviors: it retrieves the data from the database shared 2 and sends it to the other agents; and inserts the data received from the other agents in their respective shared database.

This method allows the communication manager to obtain a batch with all the data, without having to iterate through it, since all data that requires to be shared is localized under the same database. It also works straight forward for the remote agents when it receives a packet from them. Considering that the communication manager receives a

packet containing the batch of data from agent 4 (the identifier of the agent is one of the fields in the packet), it will just have to insert the complete batch in the database shared 4 without having to run through the whole data.

The only step that is now applied by the communication manager is the compression of the data before sending it through the network, and the decompression when it receives data from the network. There is nothing required to do related with serialization, since the communication manager will retrieve data that is already serialized. On the counterpart that receives the data, it will simply have to decompress the data and insert it into the database without any deserialization. Therefore, the communication manager does not need to know what items are being stored or their contents, it will only compress or decompress.

4.5.2 Data item lifetime

A data item's lifetime is the time in milliseconds since an item has been inserted into the RtDB2 until it gets retrieved, and it is also valid between remote agents. The strategy used to create a lifetime associated to a data item is the same used by the RtDB, as explained in section 3.5. The only difference is that the communication manager uses a `GetBatch` and `PutBatch` function, instead of a simple `Get` or `Put`; it means that the `GetBatch` and `PutBatch` needs to run through the batch to process the lifetime of each data item, like RtDB does but without a concept of batch. As mentioned before, this batch is used to grant that the whole operation is done within a single transaction.

In figure 4.5, it is possible to visualize the process that the data suffers when it is shared between two agents.

The process of generating the lifetime can be described as follows:

1. Internal process inserts a structure into the storage using `Put`. The method will serialize the structure received and prepend the current timestamp to it.

2. Communication manager obtains all the structures in the shared database using `GetBatch` that will subtract all timestamps to the current one, obtaining the time that the item has been in the storage of that agent. That time will be prepended to the serialized value again. At last, the batch is compressed and sent through the network.

3. A remote communication manager will receive the batch and decompress it. After decompressing, it will add the current timestamp to the time prepended in step 2 to register the moment it was retrieved, and then stores it. This operation is completely covered by `PutBatch`. This method actually adds an additional time besides the timestamp, that is the average time that the batch takes to go from one agent to another in the network.

4. Later, a process will retrieve that item using `Get`. It will subtract the stored timestamp to the current timestamp to discover its lifetime, and deserialize the structure.

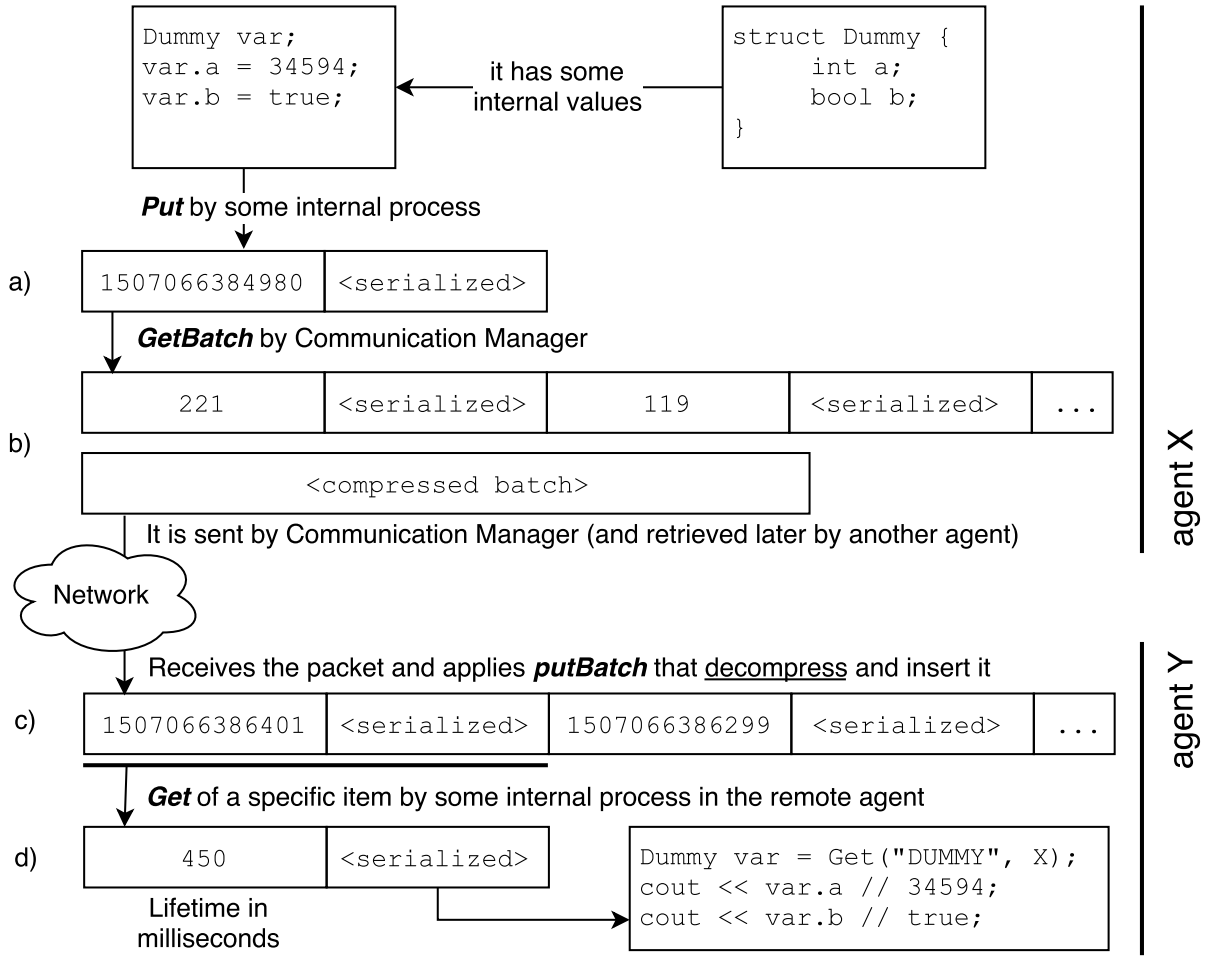


Figure 4.5: This image represents the generation of a lifetime between two remote agents applying the same process as the RtDB, with the difference that the data gets serialized and compressed. The figure contains four notes that have the following meaning: a) A timestamp (e.g. 1507066384980) is generated on `Put` and prepended to the structure; b) The stored timestamp is decremented to the current one (e.g. $1507066385201 - 1507066384980 = 221\ ms$) and prepended again for every item in the batch. After it, the batch is compressed serialized; c) Subtracts the differential value to the current timestamp (e.g. $current\_timestamp - 221$) and prepends to the structure for each data item in the batch received; d) Substracts the stored timestamp to the actual one, obtaining the lifetime (e.g. $1507066386401 - current\_timestamp = 450\ ms$).

4.5.3 Periodicity and Phase

In the RtDB2, there is a feature that allows to save some bandwidth when sharing the items over the network. Each data item has a boolean associated that indicates if an item is shared or local, but besides that there are two more fields: periodicity and phase.

The periodicity of an item is the time period at which an item is sent by the communication manager through the network. It means that if the periodicity is 1, that item will be sent every cycle. However, the periodicity might be any other integer bigger than zero,

such as five and it means that the communication manager will send that item every five cycles instead of every cycle.

The phase of an item is a required configuration to the item that allows to properly schedule how the items are sent. In this case, the default phase is zero, meaning that an item will be sent on the first cycle of the communication manager. However, if it is defined to three, it will force the communication manager to skip the first three cycles for that item before sending it.

It is possible to benefit from this feature by sending the items alternately. For instance, considering a situation with two data items being shared with a periodicity of 2, and one of them with a phase of 0 and the other one with a phase of 1. As a consequence, both items are only sent every two cycles, but the phase will force them to be sent in different cycles, alternately.

The implementation for this feature is a counter in the communication manager and two additional fields associated to each item in the configuration file (mentioned in section 4.5.4).

4.5.4 Configuration File

In RtDB, the configuration file is large and requires another tool to generate two files that are used directly by the source code. The initial configuration file is never used in the source code, it is only used to generate two files: `rtdb.ini`, which is the file loaded by the RtDB that will let it know the space that is going to be used by the structures; and `rtdb_user.h`, which is the header file that contains several macros with the items names, making the usage of the RtDB more user-friendly.

The scenario in the RtDB2 is fairly different. There is a configuration file that might be used alongside with the RtDB2, although it is not completely required. Listing 4.1 shows an example of this configuration file.

Listing 4.1: File used to configure the RtDB2 internal storage (rtdb2_configuration.xml)

```
<!-- This file can be generated using xrtdb to grant
     backwards compatibility between RtDB and RtDB2 -->
<!-- Every key might have the following parameters:
     * id that stands for a string identifier;
     * shared that indicates if the object is shared or
       local to real-time database;
     * oid (optional) corresponds to an older identifier
       used in the RtDB (this field should disappear after
       a complete upgrade.
     * period (optional) and phase (optional) in order to
       schedule how an item is shared by the Communication
       Manager -->

<RtDB2Configuration>
  <General>
    <DefaultKeyValue shared="true" period="1" phase="0"/>
    <Compressor name="zstd" dictionary="true"/>
```

```
  </General>
  <Keys>
    <key id="COACHMAP" shared="false" oid="6"/>
    <key id="COACH_INFO" oid="2"/>
    <key id="FORMATION_INFO" oid="4" phase="1" period="2"/>
    <key id="GRIDVIEW" shared="false" oid="5"/>
    <key id="LAPTOP_INFO" oid="1" phase="1" period="2"/>
    <key id="MONITOR_RTDB" oid="7"/>
    <key id="ROBOT_WS" oid="0" period="2"/>
    <key id="TEAMPLAY" shared="false" oid="8"/>
    <key id="VISION_INFO" shared="false" oid="3"/>
  </Keys>
</RtDB2Configuration>
```

The configuration file uses a XML format, instead of having a custom one. It is composed of two different sections:

- General section that only contain two different nodes:
  - Description of the default key behavior. This node contains the default values for each attribute when they are missing or when the key is being introduced for the first time. Each key can declare if the item should be shared with other remote agents or not, and also periodicity and phase of that item when being shared.
  - Compressor used by the RtDB2. In this case, only two were compiled and built in with the RtDB2, `lz4` and `zstd`. These were the ones tested and already used, although more can be inserted into the system.
- Keys section containing a list of keys. The keys need only be declared in the configuration file if its behavior differs from the one presented in the `DefaultKeyValue` from the general section. Each key must be identified by a `string` identifier and can also have four optional attributes: "shared", "period", "phase", and "oid". The first three attributes are exactly the same as the ones mentioned in `DefaultKeyValue`, but with possible different values. The attribute "oid" is used to grant backward compatibility with the RtDB2 for someone that keeps using the old method of generating the configuration file in the previous RtDB. However, it should be deprecated later with the complete replacement of the RtDB2. The backward compatibility with the previous system is explained in section 4.6.3.

When a key is missing from the configuration file, it can still be inserted and retrieved, and even shared. It can easily be inserted using the API that allows the insertion of a `string` key that was not previously inserted, assuming the default key-values expressed by the configuration file. If the configuration file is missing, the RtDB2 will assume the default values defined in the source code, meaning that all keys would be shared with a periodicity of one and phase of zero.

Moreover, it is possible to completely discard the configuration file and always use the default values defined in the source code, although it might not be the best and desirable

solution, since all the items would be shared and some of them might be fairly big to be shared over the network.

4.6 Implementation

This section is more technical than the others. It includes the API, decisions that were made in the source code in order to improve the usability or to implement a specific feature, and some features that were added.

The software structure of the RtDB2 has some differences when comparing with the older implementation of the RtDB. Its interface is now implemented with `C++` instead of `C`, although it still has a usable adapter that allows to use it in `C` without modifying any invocations to the API from the older RtDB. This backward compatibility with the RtDB is explained later in section 4.6.3.

The RtDB2 keeps all the complexity of the code away from a programmer using the API, giving some abstraction. The programmer does not know what happens to the data when he inserts an item in the database, or how the timestamps associated to the data structures are generated.

4.6.1 Application programming interface

The RtDB2 API is similar to the previous RtDB, even though one is implemented in `C++` and the other in `C`.

Since this API is done in `C++`, it has a constructor and a destructor that simplifies the use of this storage. In RtDB, it was required to invoke `DB_init` in order to create the storage, and `DB_free` to release the memory allocated. In the RtDB2, it is impossible to invoke a `Put/Get` without initializing the database, because it is associated to the constructor and the memory is always released correctly with the destructor. There are two available constructors:

1. `RtDB2(int db_identifier)` initializes the internal structure of the RtDB2 with the specific database identifier specified just to distinguish from other instances when sharing items. The storage is initialized in `/tmp/rtdb2_storage`, although it is only created when there are new items, in order to prevent empty databases in the filesystem. The default goes to `/tmp`, so that it can be discarded by the system when it gets powered off. The RtDB2 requires a location on the disk because LMDB uses memory-mapped files, as mentioned in section 2.3.1.

2. `RtDB2(int db_identifier, string path)` initializes the storage equally to the previously constructor, but instead of spawning the storage at `/tmp/rtdb2_storage`, it will spawn at the given path.

Other feature in this API is the consistency of the return types. Any function whose return value is of `integer` type will always return an error code, instead of a mixture

of timestamp with error codes, as before. The timestamp is always passed through a parameter when not pointing to `NULL`.

The API no longer uses `integer` keys internally; it now works with `string` keys that are more flexible for the programmer and it has no impact to the storage database, in this case, `LMDB`. However, the `integer` keys used are still viable and acceptable by the new API to support the backward compatibility with the RtDB.

The public API is composed by a set of nine functions, three of them with a version for backward compatibility. In the following functions enumeration from the API, considers that the type `T` stands for a template that accepts any class or data. Therefore, the API is composed of the following functions:

1. `int get(string key, T* value, int& life, int db_src)`
   The method `get` allows to retrieve data under the key required in the parameters into the pointer value. The value and life parameters are supposed to be written by the API; value is fulfilled with the data structure stored or `nullptr` if it does not exist, and life is updated with the current lifetime of the item. This method is going to retrieve a value from the database specified by the identifier `db_src` through the parameters. The internal implementation will be responsible to fetch from the correct storage.

2. `int get(int key_id, T* value, int& life, int db_src)`
   This method has the same behavior as the previous function. However, it allows to specify an `integer` identifier that must have been previously declared in the configuration file under the attribute "oid", that stands for old identifier. This attribute is used to correspond the older `integer` identifier to the new `string` identifier. Therefore, this function is used to grant backward compatibility with the older version of the RtDB2 - meaning that this function should be deprecated once the RtDB2 fully replaces the older version.

3. `int get_batch(string& batch, bool exclude_local = true,`
   `bool compress = true)`
   This method allows to obtain a batch of the content of the local storage, including all its data or only the shared data. It also allows the application of compression over the data retrieved. The internal implementation fetches a vector of key-values from `LMDB` using transactions, and then serializes it. At the end, it may also apply compression if it was requested through the parameter.

4. `int put(string key, T* value)`
   The method `put` inserts the given data structure in the field value as a serialized value in the key given by the parameter - this value is inserted under the database identifier initialized by the constructor.

5. `int put(int key_id, T* value)`
   It has the same behavior as the previous method. However, the key is an `integer`

that must exist in the configuration file under the attribute "oid" - meaning that this function should be deprecated once the RtDB2 fully replaces the older version.

6. `int put_root(string key, T* value, int life, int db_dst)`
   This function inserts the given data structure in the field value as a serialized value in the key value given by the parameter, although it allows to define from which database does the data structure belongs, `db_dst`, and the current lifetime of the given item - meaning that this function should only be used in specific cases, such as inserting data received by the communication manager into the local storage.

7. `int put_root(int key, T* value, int life, int db_dst)`
   It has the same behavior as the previous method. However, the key is an `integer` that must exist in the configuration file similar to the `put` method with the `integer` key, but with the behavior from `put_root`.

8. `int put_batch(int db_identifier, const string& batch, int life, bool shared = true, bool is_compressed = true)`
   This method inserts the batch that was created by `get_batch`. It allows to specify the database identifier where the batch belongs to. It also allows the specification of a time that will be incremented to each entry from the batch to simulate delays in the network.

9. `const& RtDB2Configuration get_configuration()`
   This method allows to retrieve all the data specified in the configuration file, such as the phase of a given key. When the key was not specified in the configuration file, this class will be responsible for giving the default values without the programmer noticing that the key does not exist in the configuration file.

4.6.2 Class Diagram

A class diagram is a typical diagram used to describe the structure of a given system. In this case, the representation uses Unified Modeling Language (UML). The architecture of the RtDB2 was made with focus in flexibility, meaning that it is simple to replace the `LMDB` by another database, or the compressor, or even the serializer. The result of the class diagram can be visualized in figure 4.6.

The initial idea for the architecture was to use three interfaces, one for each tool: storage, serializer and compressor. The storage and the compressor have their correspondent interface. However, the serializer had to be defined as a static class in order to be correctly implemented. It is not possible to have virtual functions with templates parameters at the same time, since the templates are replaced by the compiler for each type that uses it and the virtual functions only pick which function to call next in run-time. Therefore, it is impossible to have virtual functions with template parameters.

The storage interface has four virtual methods: insertion of a given key, retrieval of a given key, insertion of a batch of pair key-values and retrieval of a batch key-values.

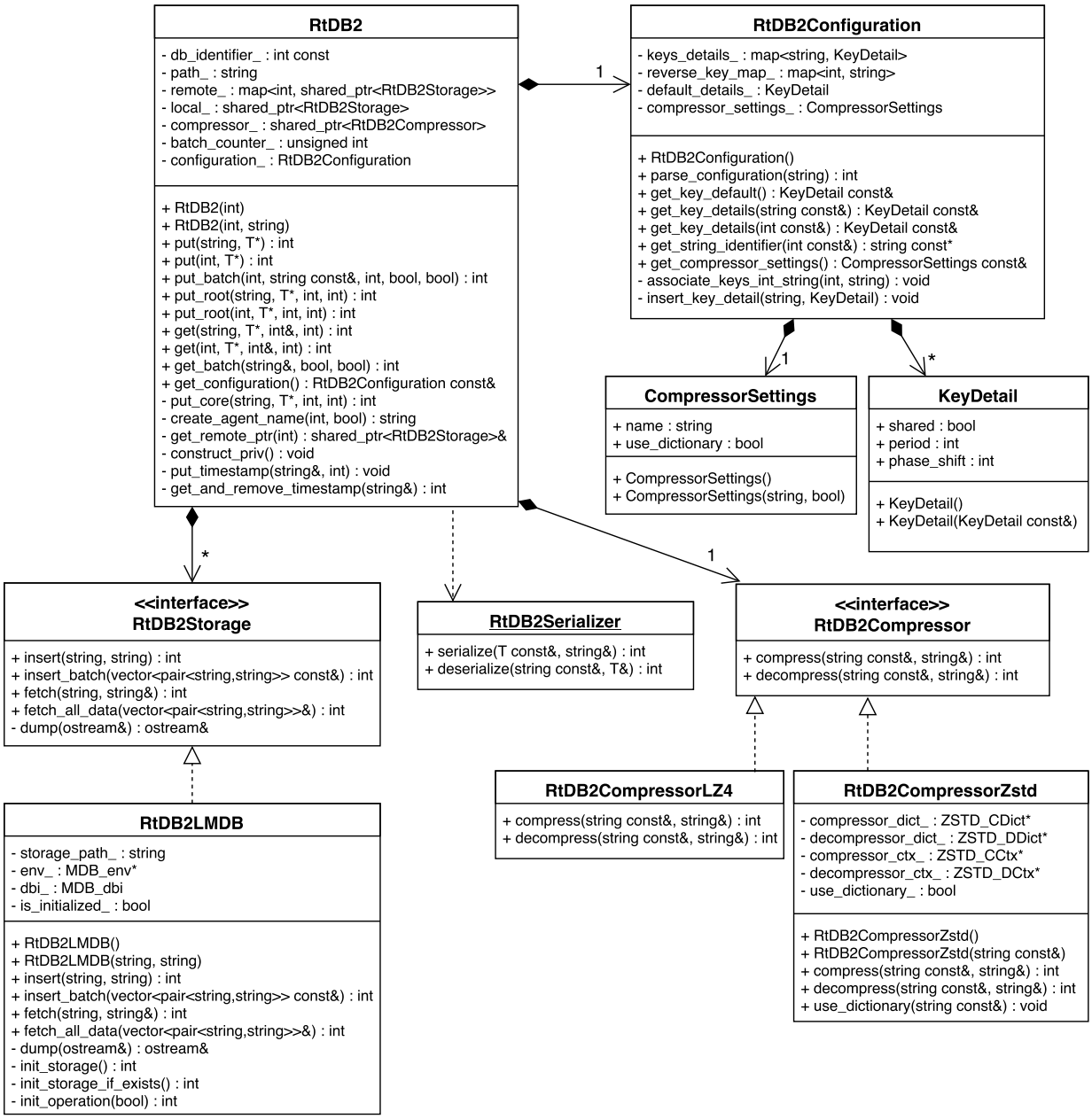


Figure 4.6: This figure represents the class diagram of the RtDB2 architecture. It has a main and public API that is accessed by the class RtDB2. Each other classes in the diagram help to make this API more complete. Therefore, there are more four important classes that the RtDB2 depends on: `RtDB2Storage`, `RtDB2Storage`, `RtDB2Storage` and `RtDB2Configuration`.

The methods that retrieve or insert batches were created because some databases have the possibility to iterate through all the data in a single transaction and in an efficient way. Therefore, these mechanisms are useful when retrieving all data to send over the network. The `LMDB` database, `RtDB2LMDB`, is one example of possible implementations for

the interface `RtDB2Storage`.

The compressor interface has two simple methods as predicable: one to compress and another to decompress the data. As shown on the diagram, there are two implementations of this interface: `RtDB2CompressorLZ4` and `RtDB2CompressorZstd`. The first corresponds to `lz4` compressor and the other one corresponds to `zstd` compressor, as the names indicate. It is easy to switch between compressors by simply changing the configuration file, without having to compile any source code. By default, `zstd` is used as the compressor when required. The class `RtDB2CompressorZstd` has the option of allowing it to be used with a dictionary.

At last, the `RtDB2Configuration` class contains a description of the configuration file mentioned in section 4.5.4. It allows to the details about a given key: if it is a shared value or not; its period; and phase shift. It also efficiently supports the older RtDB identifier by having a map that corresponds the integer identifiers to the new string identifiers.

In addition to those classes, there is also an enumeration that is used by all classes in figure 4.6, known as `RtDB2ErrorCode`. In figure 4.7, it is possible to visualize all the enumerated values. This enumeration is used to represent error codes.

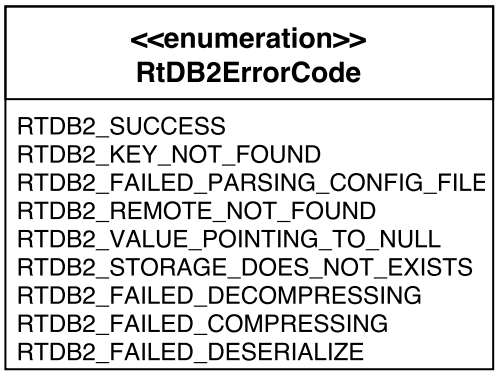


Figure 4.7: This diagram corresponds to the enumeration RtDB2ErrorCode that is used to represent all the possible error codes.

All the functions that return an integer in the RtDB2 are actually returning a value from that enumeration, thus an error code. The previous version of the RtDB used integer returns as well, but it returned negative values for error codes and positive values for lifetime. It could cause some confusion on its usage, so the RtDB2 only uses return values to give the user an error code, and lifetimes are always passed by reference through an argument.

4.6.3 Backward Compatibility with RtDB

The RtDB2 grants backward compatibility with the RtDB without requiring any additional actions. This backward compatibility also has the possibility to revert to the RtDB

with a switch of a variable named "RTDB2" in the CMakeLists.txt[6], as shown in listing 4.2, to reduce the number of actions taken by the developer to switch between both versions. Therefore, when the switch is enabled or disabled, a macro with the name RTDB2 is defined and visible through all the source code. This step of enabling or disabling was done as shown in listing 4.2, making it possible to switch back to the RtDB by changing "ON" to "OFF" on the command SET.

Listing 4.2: Code snippet from CAMBADA's project CMakeLists.txt

```
SET(RTDB2 ON)
MESSAGE(STATUS "RtDB is " ${RTDB2})
IF (RTDB2)
    ADD_DEFINITIONS(-DRTDB2)
ENDIF()
```

The previous step only disables or enables a macro, but it is not enough to grant the backward compatibility. Moreover, the RtDB2 can not ignore the older interface from the RtDB in order to grant compatibility. It must implement the functions from the RtDB through an adapter using the RtDB2 functions. In listing 4.3, it is possible to verify that the signatures from the older API are ignored and the ones from the adapter are included.

Listing 4.3: Code snippet from rtdb_api.h that belongs to the RtDB library

```
#ifdef RTDB2
   // Signatures from the RtDB2 that are compatible with the older ones
   #include "rtdb2_adapter.h"
#else
   // Signatures from the RtDB
   int DB_init (void);
   void DB_free (void);
   int DBput (int _id, void *_value);
   // More signatures ...
#endif
```

Since the RtDB2 uses templates to receive a structure instead of a pointer to void, it had to redeclare the signatures from the RtDB API. In listing 4.4, it is possible to verify that the function DB_put, that was previously declared (in the RtDB) to receive a pointer to void, now receives a template pointer. The void pointer treats the received structure as an unknown type, but it is different with templates. Templates can be used to distinguish types, since the compiler generates the code for each type that uses that function. This is required by MsgPack, because each type has a macro indicating the fields that are required to be serialized and without templates it is impossible to know which type is being serialized. Otherwise, MsgPack does not know the fields that need to be picked in order to serialize.

[6]This file is used by a build system tool, known as CMake, that contains a set of instructions describing the source code. In this case, CAMBADA is using CMake to build the source code. Future users that do not use CMake must adapt their solution to create the macro RTDB2 in order to have backwards compatibility.

Listing 4.4: Code snippet from rtdb2_adapter.h

```
#include "libs/rtdb2/RtDB2.h"

int DB_init(void);
void DB_free(void);
template<typename T>
int DB_put(int _id, T *_value);
// More signatures ...
```

Each of the functions in the adapter has a proper implementation using the RtDB2 object with its functions. In addition to the adapter, there is another required step to grant the compatibility. The RtDB uses integer identifiers for each key that are generated by `xrtdb`, although the RtDB2 uses string keys so that it can be more flexible.

As mentioned in section 4.5.4, each key declared in the configuration file may have an attribute named "oid" that stands for old identifier. It is used to allow the RtDB2 to instantly switch to the key's name that is a string when it receives an integer through the API: it is done through a map of integer keys to string values, so when the API receives an integer key, it can convert it to a string key. Therefore, the RtDB2 API has always an alternative function to insert and retrieve data that accepts an integer key, instead of string. Its implementation simply accesses the map, obtains the corresponding string key (that was assigned in the configuration file) and then invokes the corresponding function, passing it the string key.

The "oid" attribute is created through a small modification on the `xrtdb` executable in order to generate the `rtdb2_configuration.xml`, so the developers can still use the older method until they feel comfortable to completely switch to the RtDB2. This also allows to have distinct steps in the process of removing the RtDB from the source code, instead of having a bigger modification.

4.7 Tools

RtDB2 was designed keeping in mind the possibility of easily develop new tools to interact with it. The following sections present two tools that were created to simplify the usage of the RtDB2. As mentioned in section 4.2, there are two possible methods to interact with the data going through the RtDB. In this case, both tools pick the data directly from the database and deserialize it. However, it is also possible to intercept the batches of data being shared through the network, decompress and deserialize them.

4.7.1 RtDB2 Data Watcher (rtdb2top)

The RtDB2 Data Watcher, also known as rtdb2top, is a tool that allows to inspect the content of the storage. The tool was developed in `Python`, showing that is is possible to develop tools for the RtDB2 using different programming languages. It uses the `Python` module to manipulate `curses` API, which is a library that allows to develop text-based

user interfaces. The access to the RtDB2 is done through the existing `Python` API for the storage `LMDB` and the serializer `MsgPack`.

All the data inside the RtDB2 is serialized and stored into the `LMDB`, so it is not possible to view the data in a user-friendly way without deserializing the data. This tool loads the data from the `LMDB` storage and deserializes it; this means that all data can be visualized in a user-friendly interface, as shown in figure 4.8.

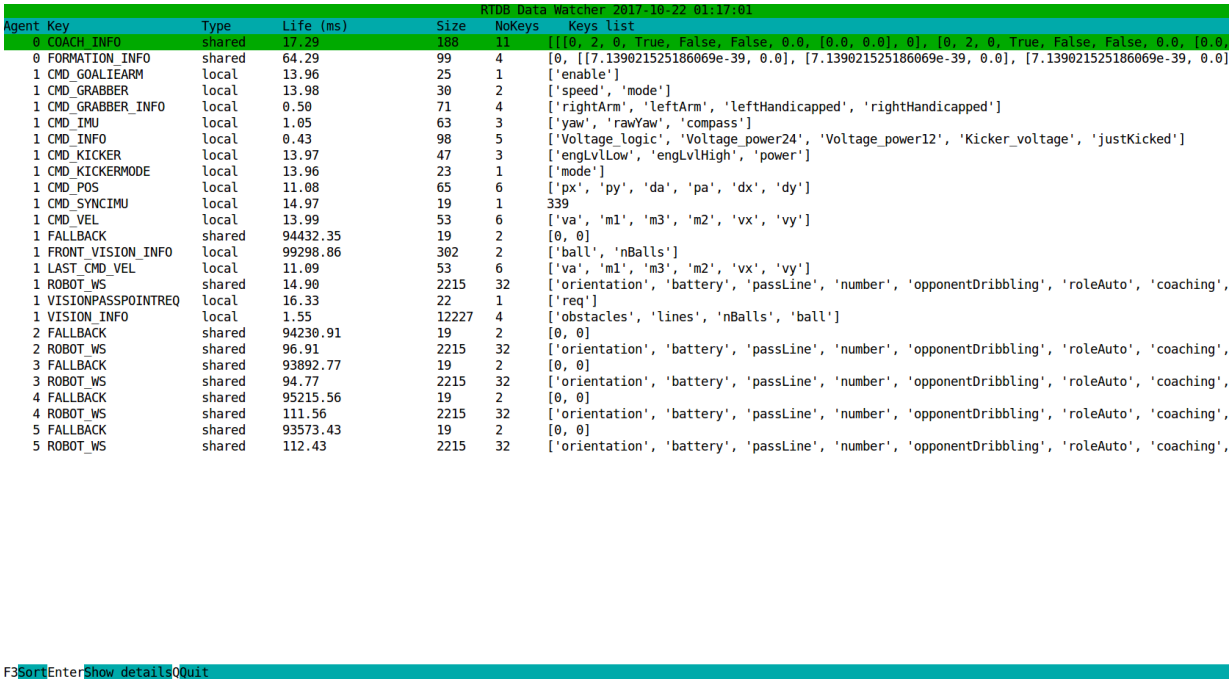


Figure 4.8: RtDB2 Data Watcher showing all the data stored. The data is represented under a table with seven columns: agent number, corresponding to the agent that created the item; name of the key; type of the item (shared or local); life in milliseconds; size of the value; number of internal fields; and a list of the internal fields names. At the bottom, it is shown the list of possible actions using the keyboard: sort the data; show more details about a given item; or quit the program.

In figure 4.8, it is possible to verify that the data is organized in a table. Each row represents an item and columns represent the data associated to it. As mentioned in section 4.5.1, the RtDB2 is composed of several instances of `LMDB` storages, thus this tool loads all of them and represents it in the columns from where the data item was fetched from; it means that each data item must have an agent and a type that are only known when the the tool picks a specific storage.

The main interface of the Graphical User Interface (GUI) allows to sort the data in the panel by any of the columns, except when using the list of keys. Moreover, the data is sorted with the agent numbers from where those items belong by default, but it can be easily sorted by the size of the data. The sorting is done by a preference of the columns. In figure 4.8, the items are sorted with the default configuration, so they are sorted by the

following preference of columns: “Agent”, “Key”, “Type”, “Life”, “Size”, and “NoKeys”.

Besides the sorting feature, it is also possible to represent the details of each items by hitting the “Enter” key or exit the program by pressing “Q”.

Figure 4.9 shows the result of pressing “Enter” in a given data item. It takes a snapshot of that item and shows through simply panel in `JSON` format. The representation of the item may also contain internal structures, by representing them recursively. As an example, figure 4.9 contains one field named “passLine” shown as a vector of two points, which is a simple structure with two fields, `cambada::geom::Line`, that also gets serialized. That structure was set to not be mapped by `MsgPack`, so the field names do not appear in that vector.

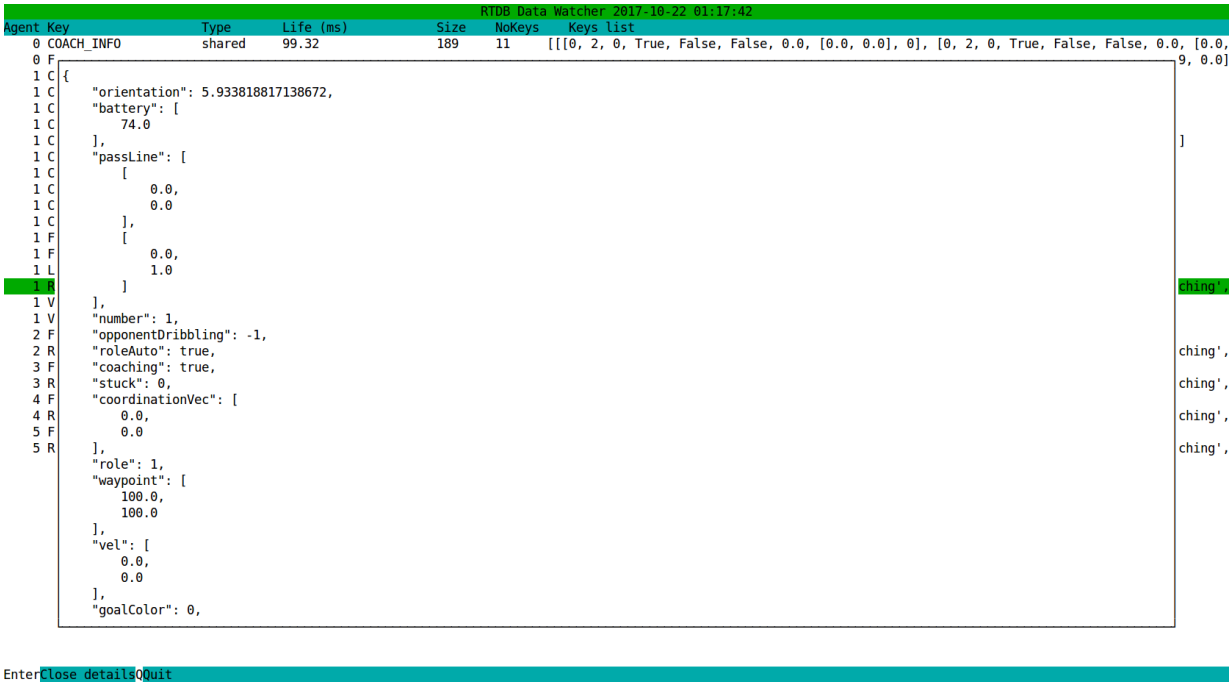


Figure 4.9: RtDB2 GUI showing the details about the item `ROBOT_WS` with their values and its inner structures

4.7.2 Dictionary Generator

This tool is responsible for generating dictionaries for `zstd`. As mentioned in section 4.4.3, `zstd` might be used as a simple compressor without any previous knowledge, although it is also possible to use it with previous knowledge. The compressor accepts a dictionary when it is being opened, and that dictionary will contain previous knowledge. Moreover, the dictionary can be trained with the typical data that is going to be compressed in order to improve the results. A dictionary helps to improve the final size of the compressed data and the performance while compressing and decompressing.

This tool does five distinct steps:

1. Initializes the RtDB2 and inserts all possible items into it under the same agent identifier - the structures are initialized with zeros.

2. Fills in the items with random values - this way, the dictionary is not biased by the values, but only from the strings keys resulted from the MsgPack mapping. It is not desired that the dictionary gets biased by values that are not real.

3. Obtains a batch from the current state of the RtDB2 storage (as communication manager does) and stores it into a folder.

4. Repeats steps 2 and 3 for `N` iterations (an input parameter).

5. At last, the stored batches are used to train the dictionary using a tool provided by `zstd` for training.

To summarize, the idea is to allow generate a dictionary without having to collect data for several minutes under different situations and still obtain a good compression. The `MsgPack` serializes everything before the data gets to the compression, meaning that there are three things that get serialized: keys for the data; field names in the structures; and values. These field names are going to appear repeated among several structures and the keys are always the same. This tool allows training the dictionary to detect most of these situations by randomizing only the values in the data structures, meaning that the keys and the field names are always the same. This situation tries to simulate what happens during a match, the keys and the field names never change, only the values will be constantly changing.

The step that generates random data into the data structure was a necessary step to prevent the trained dictionary from overfit. In other words, overfit means that the trained model is too close to a particular data set, typically the one used to train. The random data will prevent overfitting in the dictionary by varying the values in each data structure field. This tool also allows to train the dictionary using real data that is received by an agent during its execution, instead of generating values to fill up the structures.

4.8 Summary

This chapter presented the RtDB2, which is the solution designed to improve the already existing system, known as RtDB. Similar to the previous solution, it is based on a Blackboard architecture, meaning that all the data can easily be retrieved by any internal process. Moreover, it also uses the communication manager to share the information among the several instances of the system. However, the strategies used to store, retrieve, send or receive data are different.

The RtDB2 no longer stores an object by copying its memory address space into a pre-allocated space reserved for the system, as RtDB did. Instead, it now uses two tools

for this process, a serializer (`MsgPack`) and a storage (`LMDB`). Therefore, when inserting an object, it will first serialize the object and then store it into the storage. On the other hand, when retrieving an object, the RtDB2 will fetch the data from the storage, deserialize it and return it. This method is more complex than the one used by the RtDB, but it also brings many advantages with it. Besides the strategy used to store and retrieve items locally, the replication has also changed. The communication manager fetches the data items from the storage as a batch using transactions; it means that the batch contains only data from the same moment. Moreover, it will also apply compression (with `zstd`) before sending the data over the network to reduce the bandwidth usage as much as possible.

Throughout this chapter, it was shown the requirements and strategies used to bring improvements to the previous solution. As a result from these decisions, there are several advantages that the RtDB2 has brought when compared with the RtDB, such as:

- Support for dynamic data structures, such as `std::vector` from `C++`.
- Independency from the architecture of the machine, since they are being serialized.
- Insertion and retrieval of data items that were never declared before in the configuration file.
- Ease of use, as it does not require to generate configuration files or to grant the same configuration over the existing instances.
- Possibility of compressing the data when sending it over the network.
- Possibility of creation of new tools to incorporate with the RtDB that can work in other programming languages and externally to the system. An example of this feature is the already existing tool known as `rtdb2top`, which permits to inspect the content of the storage of a given instance.
- The system can now keep the data when the system crashes or restarts.

It is known that the RtDB2 has more complexity internally, although it also has many features that bring flexibility to the system, as described through this chapter. Therefore, this solution is more likely to have a decrease on the performance when compared with the RtDB, since it is a more complex solution. On the other hand, it also has many advantages that allows it to be a viable solution, as long as the impact on the performance is negligible.

Chapter 5

Experiments and Results

This chapter starts by explaining the experimental setup used to ensure that these tests were fairly compared agaisnt each other, with the respective hardware specifications. Throughout this chapter, it is possible to find out several features that required to be tested. It includes a comparison between the performance of the two most used functions, `Get` and `Put`, against the ones from the RtDB. There is also a comparison between the new method that allows to obtain a batch of information and the older method used by the RtDB to retrieve all data.

Besides the performance achieved by each method, some tests were done to make sure that the bandwidth that is going to be used is still under the limits imposed by the MSL. Therefore, the maximum bandwidth that can be used by each team is 2.2 Megabits/second[1]. It means that the existing comparisons are not only regarding response times, but it also compares the size of the data.

For this tests, it is important to state that the agent's cycle is executed every twenty seconds, the only important deadline. The agent is a soft real-time system. It means that a deadline can be missed, although the quality of the system might degrade. For example, the agent takes thirty seconds to process its cycle, which implies that some of its actions might be applied later, such as a late movement. Another important statement is that on average, an agent does twenty insertions and forty retrievals every cycle.

5.1 Experimental Setup

All the experiments were done in the robotic soccer field, built within the IRIS Laboratory, part of IEETA. The field has the official MSL dimensions, only with an area around the delimited field a bit smaller than the one defined by the rules. Three players and the basestation were used in the experiments, being the hardware of every player equal to the one used during competitions. The laptop of every player, where the high level software runs, is composed of: 8 GiB of RAM (2x 4 GiB SODIMM DDR 1600 MHz), Intel Core i5-3340M CPU @ 2.7 GHz and Micron C400 RealSSD with 128 GB of space. Figure 5.1

[1]MSL with the respective rules: `http://www.robocup.org/leagues/6`

represents the used setup. Besides it, some obstacles were being randomly placed during the experiments and moved around to increase the size of the structures as much as possible.

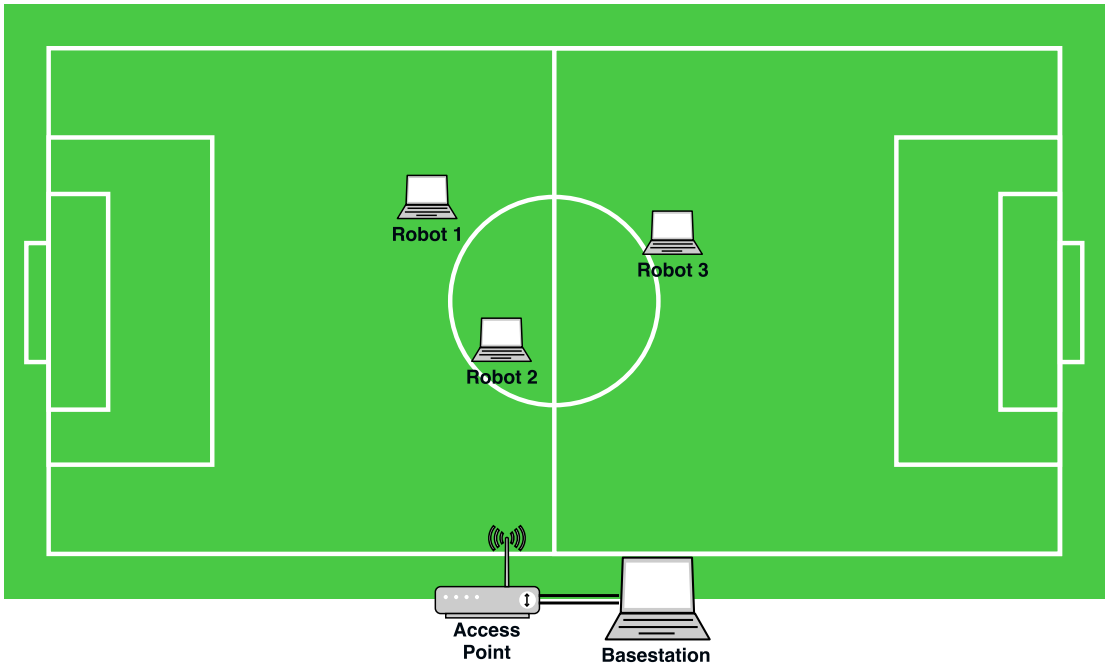


Figure 5.1: Experimental setup used in the laboratory to take the results. It uses an access point to provide communication between the devices. There is a device with the basestation role connected through Ethernet and three robots connected through wireless to the access point. The figure proportions do not correspond to the real ones.

The experiments were conducted using both the proposed RtDB2 and the existing version, RtDB, in order to make a comparative analysis of the results obtained. All data was retrieved from the same player (robot) to not bias the results. Besides the results obtained in the IRIS Lab, data was also collected during the Portuguese Robotics Open 2017 in Coimbra, where the RtDB2 was used. Later on, these packets were used to create a dictionary that is going to be shown later in this chapter.

5.2 Base operations

The first experiment was designed to evaluate operations `Get` and `Put`, which are the main operations executed by normal processes, others than the communication manager.

The behavior of the `Put` operation used by the RtDB inserts the data into the shared memory with the lifetime appended to the data. This means that the operation copies the region of memory pointed from the data structure passed by argument into the storage using `memcpy` function from `C++`. This scenario is quite different in the RtDB2, which

requires to serialize the object, append the lifetime to the result of the serialization and then insert it into the storage.

The `Get` operation has a similar behavior, but with the opposite order. In the RtDB, it will simply read from the shared memory of the desired object. In the RtDB2, it is required to retrieve the object from the storage, extract the lifetime from that object and then deserialize it.

Some tests were done by running the robots with the RtDB active while recording the time taken by each operation and the size consumed by the data structures. On a second part, the source code was recompiled to be executed with the RtDB2 and then retake the times in order to be able to compare them. For these operations, the size of the raw data structures and the size of the data after being serialized was also recorded.

5.2.1 Overall comparison with RtDB

Comparison of elapsed times

The wall time is the time since the method has started until it has finished, also know as elapsed time. In table 5.1, it is possible to find out some metrics related to the elapsed time taken by those operations under the RtDB and the RtDB2.

Table 5.1: Comparison between the basic operations (`Get` and `Put`) from each system, RtDB and RtDB2, in terms of elapsed time. All the values presented in this table are in microseconds. The table shows the average, standard deviation, median and maximum for each situation.

	RtDB		**RtDB2**	
	Put (μs)	**Get** (μs)	**Put** (μs)	**Get** (μs)
Average	1.697	1.736	18.581	24.123
Standard Deviation	14.904	7.283	264.834	123.736
Median	1.21	1.295	11.707	11.623
Maximum	3489.787	2222.979	105419.579	86905.579

It is clearly visible that the previous version of RtDB performs better than the RtDB2, due to the simplicity of the implementation. The RtDB simply copies the data from storage or copies it into the storage. In the RtDB2, the process is much more complex than a simple copy: it has the serialization associated with it and an actual database that is responsible for handling the requests.

The insert operation takes on average 1.697 microseconds in the RtDB and 18.581 microseconds in the RtDB2, this difference between RtDB and RtDB2 is a great impact on the response time. The same goes for the operation that retrieves data: it takes on average 1.736 microseconds in the RtDB and 24.123 microseconds in the RtDB2. However, it is important to state that this does not mean that the RtDB2 is not viable. Flexibility in general does not come without a cost. Since the times obtained are only a very small fraction of the agent's cycle (20 milliseconds), the solution is viable.

In the worst situation, the `Put` operation took 105 milliseconds to complete and the `Get` operation took 86 milliseconds, which is much higher than the average. This value

might degrade the quality of the system, since one insertion takes five times more than the deadline of the agent's cycle. However, by looking to the data, only four values (4 in 350946) are bigger than the agent's cycle and all of them belong to a process named monitor, which is a process that runs in the user-space. Therefore, those values are outliers that have no impact on the system; they belong to message that allows the basestation to show which processes are active or inactive. However, 105 milliseconds it still unnoticeable since these structures are manually checked through the basestation's GUI.

Comparison of CPU times

This section discusses the responses times measured in CPU time. This metric only counts the time that the Central Processsing Unit (CPU) spent using the method code, and not external causes, like preemption. It is important to show that these long response times are not caused directly by the RtDB2, so it is relevant to find an alternative to the elapsed time. The elapsed time is the actual time that the method has taken to execute, thus it will keep counting the time even if it gets interrupted. Therefore, in table 5.2, there is a comparison of CPU times between the RtDB2 and the RtDB.

Table 5.2: Comparison between the basic operations (`Get` and `Put`) from each system, RtDB and RtDB2. All the values presented in this table are always in microseconds. The table shows the average, standard deviant, median, maximum and sample size for each situation. The times are determined in CPU time, thus the time represented was measured using CPU cycles.

	RtDB		**RtDB2**	
	Put (μs)	**Get** (μs)	**Put** (μs)	**Get** (μs)
Average	0.861555	0.939667	15.16567	22.36069
Standard Deviation	1.299152	1.802651	14.25545	37.43912
Median	1	1	10	10
Maximum	65	78	228	336

It is possible to verify that the maximums are now much smaller than the ones found in table 5.1. Therefore, the main cause of the largest values found during the `Put` or `Get` operation are caused by the method blocking itself for some reason or the method being preempted by another process with higher priority, such as the vision process blocking monitor process or a system process blocking any other.

Assuming that it has a Gaussian distribution and using 68-95-99.7 rule, it is possible to calculate the upper bound that an operation might take. Therefore, the interval where the values will lie in 99.7% of the cases can be calculated by the following expression:

$$[\mu - 3 \times \sigma, \mu + 3 \times \sigma]$$

Where μ stands for the average and σ stands for the standard deviation. Therefore, in a `Put` or `Get` operation, the upper bound of the interval (which is the one counting for the deadline) in the RtDB2 is defined by:

$$T_{Put} = 15.16567 + 3 \times 14.25545 = 57.93202 \text{ microseconds}$$
$$T_{Get} = 22.36069 + 3 \times 37.43912 = 134.67805 \text{ microseconds}$$

To conclude and taking into consideration the initial statements, the `Put` operation in the worst situation (upper bound) of the twenty insertions in a specific cycle will take $20 \times 57.93202 = 1158.6404$ microseconds. The `Get` operation will consume with its forty retrievals a maximum of $40 \times 134.67805 = 5387.122$ microseconds. Moreover, a total of $6545.7624\ \mu s \approx 6.5\ ms$ from a total of $20\ ms$. It consumes almost one third of the complete cycle from the agent considering the upper bound, although some factors were ignored, such as the possibility to operate concurrently, which is very likely to happen. Moreover, the worst situation is very unlikely to happen, since it is considering the worst case in all the insertions and retrievals. The processing of the agent is quite viable in this situation even though it has only two third of the complete cycle.

Frequency of the elapsed time

In figure 5.2, it is possible to find out the most frequent time that the `Put` operation takes to finish, since it is an histogram of the time elapsed by the operation. Therefore, it helps to understand its behaviour.

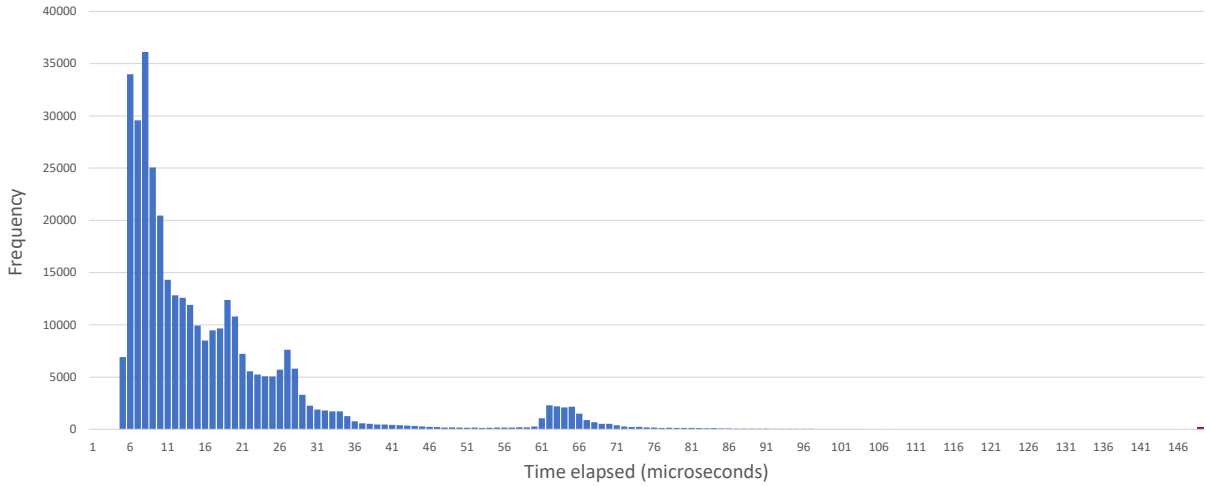


Figure 5.2: This figure shows the histogram of the time taken during a `Put` operation. It allows to visualize the typical time that the operation takes and how many occurrences can be found for a specific time. The vertical axis contains the number of occurrences and the horizontal axis represents the time elapsed during the operation. The total number of operations done was 350946. Occurrences with bigger time elapsed than 146 microseconds were not represented in order to be possible to visualize the curve.

Figure 5.2 shows the average time taken by a `Put` operation, 18.581 microseconds, that corresponds to where most of the data is in the histogram. Besides the maximum found in the histogram at 8 microseconds, there are several local maximums at around 18, 27

and 63. These local maximums are caused by the key that is being stored. The operation does not have a constant cost, it tends to vary for each key that is being stored. As an example, the local maximum at 63 is caused by the `VISION_KEY` structure that is being stored, it usually takes on average 63.69 microseconds (as shown in next section), which is the largest data structure. Therefore, it is important to analyze the cost taken by each operation, for each key, to understand which keys cost the most.

5.2.2 Operations cost grouped by keys

It is very likely that the time taken by each operation varies depending on the key that is being retrieved or inserted, since the size of the keys might vary a lot and it has a great impact when comparing performances. Before showing the comparison among the performance for each item, it is important to verify the size of the data for each item and its serialized size on average, as show on table 5.3.

Table 5.3: Size of the data for each key and the respective average of the serialized size. All the values are represented in bytes.

Key	Size (bytes)	Serialized Average (bytes)	Key	Size (bytes)	Serialized Average (bytes)
CMD_DELTATIME	4	5	CMD_POS	24	49
CMD_GOALIEARM	1	9	CMD_ROTANDKICK	20	71
CMD_GRABBER	2	14	CMD_SYNCIMU	4	1.8
CMD_GRABBER_INFO	12	55	CMD_VEL	24	37.1
CMD_GYRO	12	34	COACH_INFO	260	172.9
CMD_IMU	24	47	LAST_CMD_VEL	24	41
CMD_INFO	10	86	ROBOT_WS	608	2199.4
CMD_KICKER	3	31	VISION_GAIN_INFO	4	7
CMD_KICKERMODE	1	7	VISION_INFO	9052	12212.4
CMD_MOTORTEMP	3	40	VISIONPASSPOINTREQ	1	6

The size of the data serialized is usually larger than the raw data, with the exception of `CMD_SYNCIMU` that gets smaller. Typically, the data gets larger due to the mapping done to each field in order to allow the structures to be modified between the serialization and deserialization, as explained in section 4.4.2. `ROBOT_WS` is one of the structures that is most used and modified, therefore it also contains more mapping to grant more flexibility to the programmer, which is reflected on its size that goes from 608 bytes to 2199.4 bytes, on average. Moreover, the data structures used by CAMBADA have a great variation on their size. There are small structures and larger structures, and that should be visible on the performance of the operations, since they serialize or deserialize the data.

In figure 5.3, it is possible to visualize the average of the previous test, but grouped by each key for the operation `Put` in the RtDB2. Moreover, each average shows the time taken by each of the steps done by that operation.

It is possible to take several conclusions about the RtDB2 from figure 5.3 and according with the size of each data structure:

1. The keys with the largest data structures, `VISION_INFO` and `ROBOT_WS`, are the ones

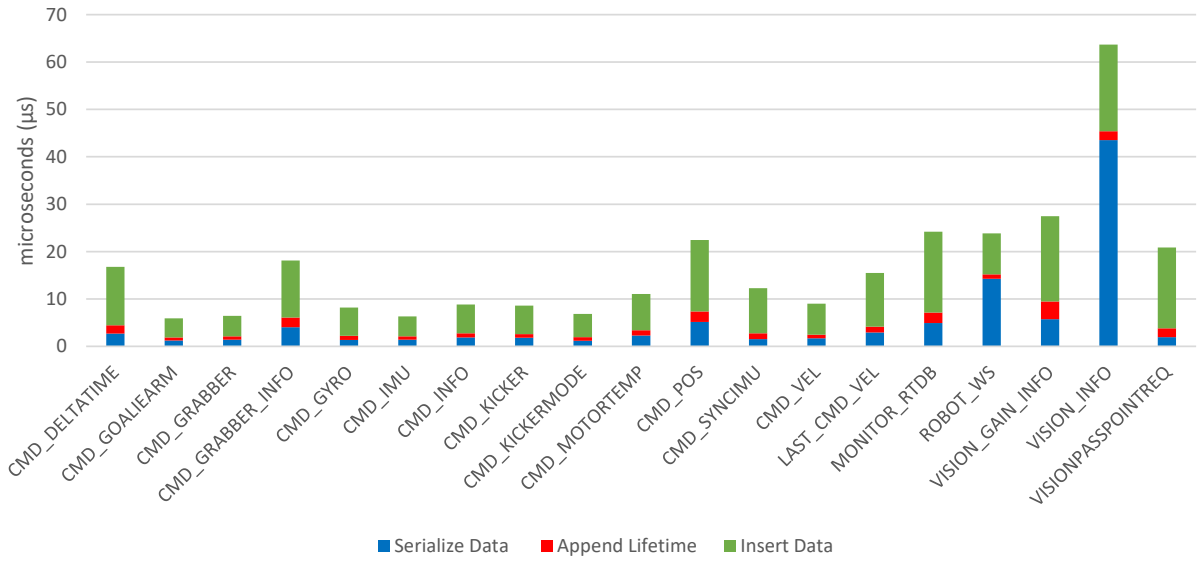


Figure 5.3: This figure shows a comparison of the time taken by each key during a `Put` operation, on average. It also allows to compare the time taken by each step internally done by the operation: serializing the object, appending lifetime and inserting the data.

---

that take more time to serialize the object, as expected. The `VISION_INFO`, which is the largest one, takes on total an average of 63.7 microseconds.

2. Appending lifetime takes almost no impact to the performance of the operation.

3. Inserting a data structure into the storage does not seem to be affected by the size of the data as much as the serialization, since the largest value, `VISION_INFO`, takes almost the same time as `CMD_POS`. However, the time taken to serialize is much different.

The standard deviation of the worst case (`VISION_INFO` key) is 14.59 microseconds. Assuming a Gaussian curve, the data in 99.7% of the cases will lie under the following interval:

$$I_{VISION\_INFO} = [\ \mu - 3 \times \sigma, \mu + 3 \times \sigma\ ]$$
$$I_{VISION\_INFO} = [\ 63.7 - 14.59 \times 3, 63.7 + 14.59 \times 3\ ] = [\ 19.93, 107.47\ ] \text{ microseconds}$$

A maximum of 107.47 microseconds has almost no impact in the cycle of the agent, since the cycles has a maximum duration of 20 milliseconds. Even if the logic from the robot takes 19 milliseconds at most, there is still time to perform several insertions of the key `VISION_INFO`. Typically, a key is only inserted once during a cycle of the agent.

In figure 5.4, it is possible to verify the operation `Get` for the same scenario. It is visible that there is an huge impact when retrieving a larger object from the storage, because it will take some time to deserialize it.

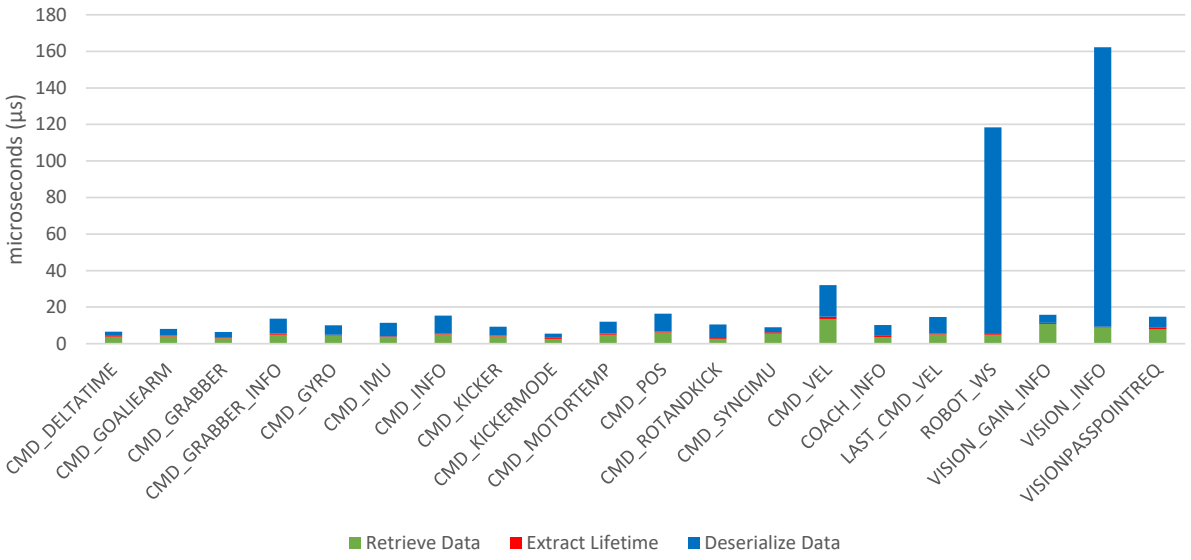


Figure 5.4: This figure shows a comparison of the time taken by each key during a `Get` operation, on average. It also allows to compare the time taken by each step internally done by the operation: retrieving the data, extracting the lifetime and deserializing the object.

The objects with the most impact, `ROBOT_WS` and `VISION_INFO`, are only retrieved once by internal processes, which is an important factor when determining the impact that such operations have. On average, the `ROBOT_WS` will take 118.3 microseconds and VISION_INFO will take 162.3 microseconds, with a standard deviation of 26.0 and 44.6 microseconds, respectively.

Assuming a Gaussian curve, the data in 99.7% of the cases will lie under the following interval, for each key:

$$I = [\ \mu - 3 \times \sigma, \mu + 3 \times \sigma\ ]$$
$$I_{ROBOT\_WS} = [\ 118.3 - 26 \times 3, 118.3 + 26 \times 3\ ] = [\ 40.3, 196.3\ ] \text{ microseconds}$$
$$I_{VISION\_INFO} = [\ 162.3 - 44.6 \times 3, 162.3 + 44.6 \times 3\ ] = [\ 28.5, 296.1\ ] \text{ microseconds}$$

Since these keys are only retrieved once and at worst case the `VISION_INFO` can take 296.1 microseconds, it does not have an impact that can be noticed or cause the agent's cycle to be skipped since it lasts for 20 milliseconds.

It is important to state that most of the time taken in both operations comes from the `MsgPack` during the serialization and deserialization. It might be interesting to try other serializers in terms of performance. However, some flexibility will be lost since the `MsgPack` was picked due to its flexibility, as explained in section 4.4.2.

5.3 Replication operations

The communication manager is responsible for sending all the data over the network and receive the one that is being sent and store it. The RtDB to replicate uses the operation

`Get` to iterate over all the existing keys and concatenate them in a buffer to send over the network, thus there is no specific operation implemented for this situation. On other hand, the RtDB2 has two specific operations: `GetBatch` and `PutBatch`. The `GetBatch` uses the concept of cursor from the `LMDB` to iterate over all the data effectively, and then compresses it. The `PutBatch` decompresses the data received and inserts it into the storage.

As mentioned before, the RtDB2 allows to easily change the compressor. In the experiment described in this section, two compressors were tested using the real data that was being received: `zstd` and `lz4`.

5.3.1 Compression Comparison

When comparing the compression tools, there are three important factors to consider: compression ratio, compression speed and decompression speed. This section shows a comparison between between the two alternatives, for the three factors.

As mentioned in section 4.4.3, `zstd` allows to use trained dictionaries to obtain better compression results. The tests shown here represent `zstd` under three distinct situations:

- No dictionary - Without any trained dictionary.
- Generated - A trained dictionary that has been generated using the dictionary generator tool that was created (tool is explained in section 4.7.2).
- Competition - A trained dictionary with data from the Robotics Open 2017 competition. That data was obtained during an official match.

Thus, the following comparisons involve four alternatives for the compression approach, a single one for `lz4` and three for `zstd`, referred to as `zstd` (No dictionary), `zstd` (Generated) and `zstd` (Competition).

Comparing compression ratios

Figure 5.5 shows the batch size after compression for the four alternatives, along side with the batch size without compression, recorded during 250 seconds. This data was obtained with the usual experimental setup mentioned in section 5.1.

During the first ten seconds approximately, there is a peak in every series that is caused by the initialization of the robot, since it does not send all the data during the initialization. Therefore, it is irrelevant for the comparison.

Later, it is possible to find out that the original data uncompressed does not vary much by staying around 2400 bytes. Since none of the series overlapped each other during the measure of the values, it is simple to indicate the performance taken by each one of them in terms of compression ratio. Therefore, `lz4` stands at last, followed by `zstd` without dictionary, as expected. In `zstd` trained with data from the competition, it compresses data more efficiently than the one with generated data; it is an expected behaviour since the generated dictionary is focused in compressing the fields names caused by the `MsgPack`.

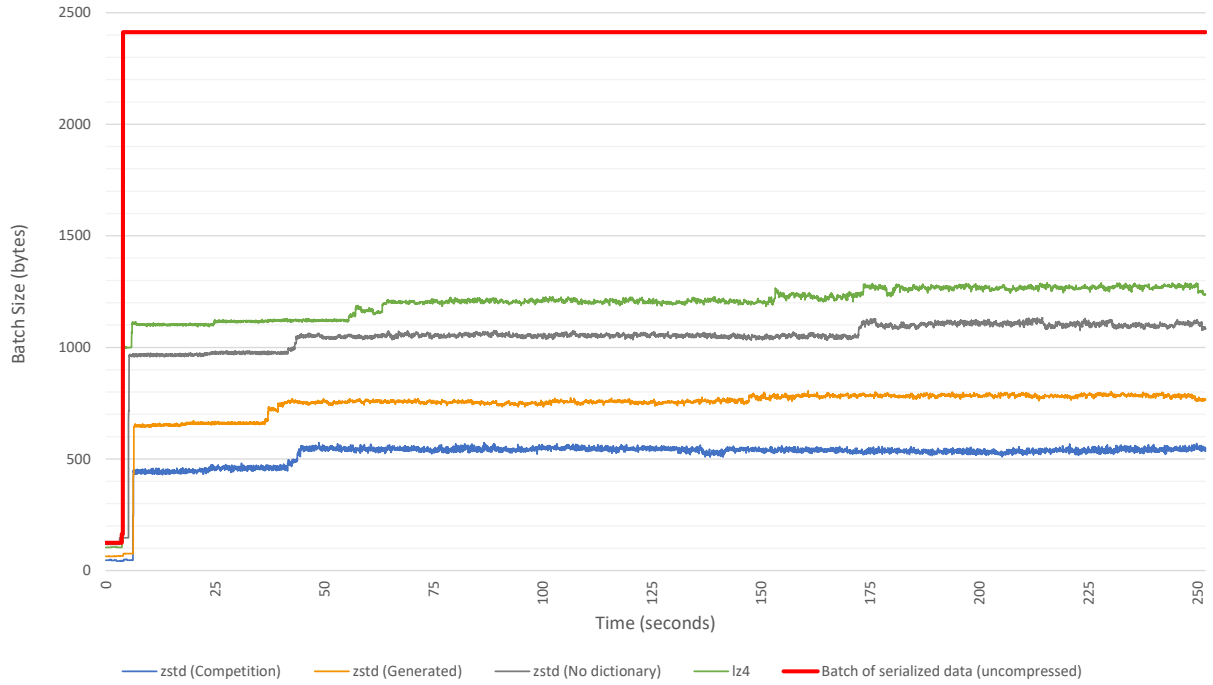


Figure 5.5: Compression Ratio comparison among several compressors. The horizontal axis represents the time since the test has started until it has ended, which took around four minutes. The vertical axis represents the data in bytes. This figure contains four compressors series and one series showing the size of the original data represented in bytes.

On other hand, the one generated from the competition takes into consideration the field names and the corresponding values by training with the most frequent ones, obtaining better results.

The table 5.4 shows a comparison of the average size for the four alternatives; this table was created with the same data as figure 5.5, but ignoring the initialization of the robots where the size is much smaller. It is possible to verify that `zstd` from the competition achieves the best compression of all four alternatives with 21.66%, which means that it reduces the size of the data almost to one fifth of the original size.

It is important to state that the dictionary generated in the competition was executed under a different scenario when tested. It was trained in Coimbra during the competition

Table 5.4: Comparison results among several compressors and their variations in terms of compression ratio and average size. The top row shows the compressors that were tested and the original data as uncompressed.

	Uncompressed	**lz4**	**zstd (No dictionary)**	**zstd (Generated)**	**zstd (Competition)**
Average Size (bytes)	2377.05	1188.53	1037.05	736.90	514.91
Average Compression Ratio (%)	100	50.00	43.63	31.00	21.66

and tested in Aveiro under completely different setup. However, it still managed to obtain the best performance, meaning that the dictionary can handle variations in the data efficiently.

Comparing compression and decompression speeds

The compression and decompression speeds were measured for the four in evaluation alternatives, using the same setup as before. The initialization values (ten seconds) were ignored, during that time the batches are much smaller and unrealistic when compared with the ones created during a match. The results obtained are depicted in figure 5.6.

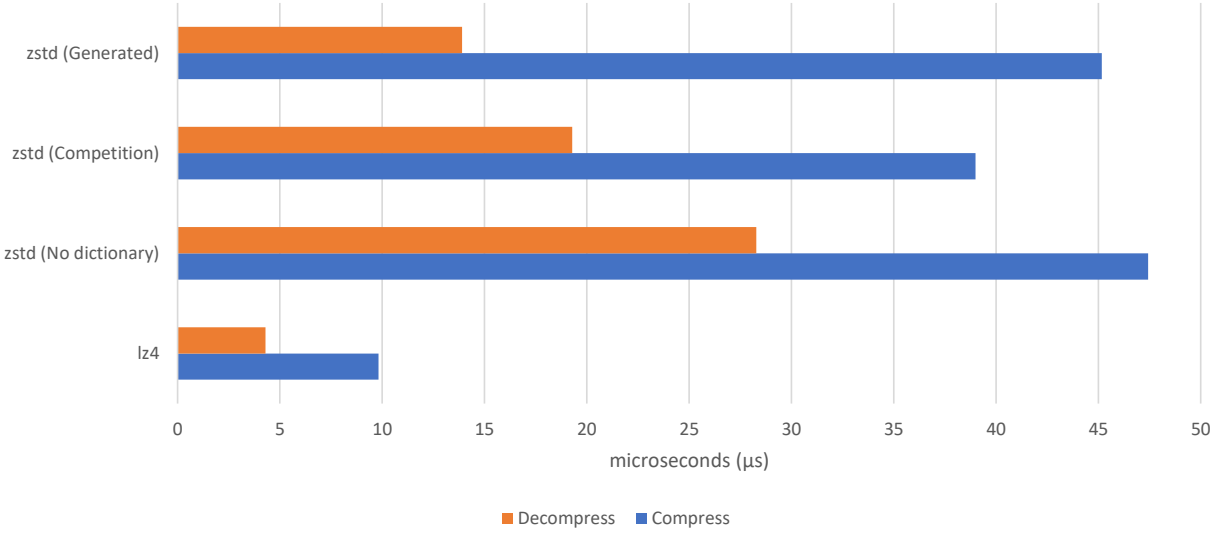


Figure 5.6: Compression and decompression speeds among the compressors. The figure represents the time taken on average by each of the compressors to compress and decompress a batch of data.

In any of the alternatives or compression approaches, the decompressing speed is higher than the compressing speed. This fact is important, since during a single cycle a robot will decompress more batches than compress. It will decompress every batch received by the other robot to insert it and compress only the batch of its data in order to send it.

It is possible to verify that `lz4` shows the best decompression and compression speeds. In contrast, in compression ratio tests it achieved the worst compression ratio when compared to `zstd` alternatives. However, it almost no influence in the choice, since compression ratio is more important than the compression or decompression speeds. As long as the decompression or compression speeds do not degrade the quality of the system, it is acceptable to give more priority to the compression ratio.

Considering the worst case, `zstd` without a dictionary, on average, is able to compress a batch in 47.43 microseconds and decompress in 19.28 microseconds with a standard deviation of 8.40 and 7.13, respectively. It means that the batch compression and decompression under the worst case are still viable, since these operations are done in a reduced amount

of times (1 compression and N decompression, where N stands for the total number of robots in the team) during an agent's cycle that takes 20 milliseconds. The compression and decompression are only part of the operation `GetBatch` and `PutBatch`, thus it is more relevant to check if the complete operation is viable or not, as shown in the following experiment.

5.3.2 Comparison with the RtDB

In this section, a comparative analysis between RtDB and RtDB2 is done, in terms of operations related with the communication manager. The topics covered are the time of retrieval of data from the storage to be sent through the network, the size of that data, and the time of insertion into the storage.

All the items to be sent in a given cycle are sent in a single message. Thus, all those items are retrieved from the storage to compose the message. In RtDB2, this done by a single function, `GetBatch`. In RtDB, this is done by a sequence of calls to function `Get`. Similarly, the insertion is done using `PutBatch` once, in case of RtDB2, or `Put` several times, in case of RtDB. In any case, the data sent to or received from the network is referred to as a batch.

Time consumed to insert and retrieve a batch

As found in the base operators, it is expected that the RtDB outperforms the RtDB2 due to its simplicity. The time consumed by by the put and the get of a batch is shown in table 5.5.

Table 5.5: Comparison between RtDB and RtDB2 for the operations that retrieve and insert a batch of data measured in CPU time. These operations were measured and presented as average, standard deviation, maximum and median. All these values are represented in microseconds. The asterisk (*) means that there is no specific operation named `PutBatch` or `GetBatch` in the RtDB, and it is done by inserting or retrieving all data using several operations `Put` or `Get`, respectively.

	RtDB		**RtDB2**	
	PutBatch* (μs)	**GetBatch*** (μs)	**PutBatch** (μs)	**GetBatch** (μs)
Average	1.61	2.07	36.97	62.54
Standard Deviation	1.51	0.97	12.42	22.68
Maximum	69	20	326	261
Median	2	2	37	55

As expected, the time consumed by the `PutBatch` or `GetBatch` is much superior to the implementation done by the RtDB. However, it does not mean that the RtDB2 is not a valid solution. The RtDB2 results must be compared with the deadline of the agent's cycle, twenty milliseconds.

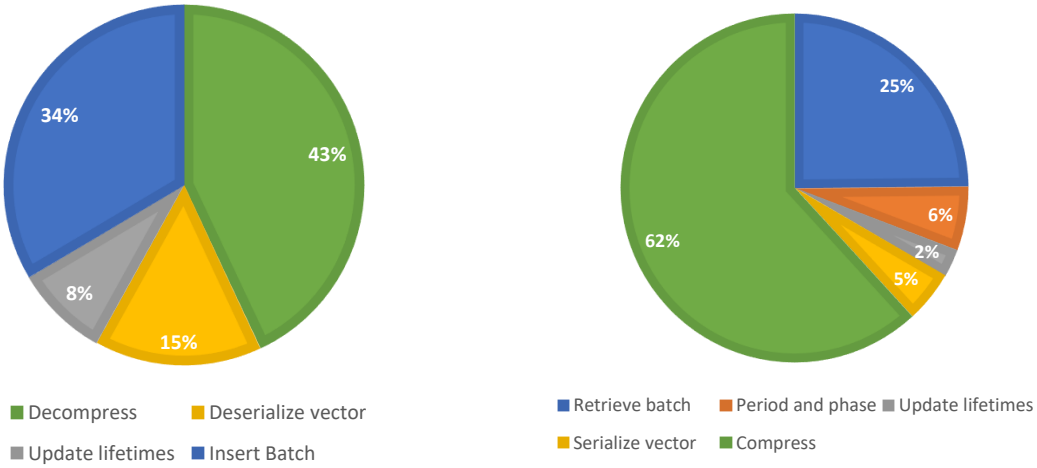
Assuming that `PutBatch` and `GetBatch` has a Gaussian curve and using the 68-95-99.7 rule, it is possible to calculate an interval where the values will lie in 99.7% of the cases.

However, only the upper bound is relevant in this situation and it can be given by the following expressions:

$$UpperBound = \mu + 3 \times \sigma$$
$$Upper_{Put\_Batch} = 39.97 + 3 \times 12.42 = 77.23 \text{ microseconds}$$
$$Upper_{Get\_Batch} = 62.54 + 3 \times 22.68 = 130.58 \text{ microseconds}$$

The operation `GetBatch` only occurs once per cycle, used by the Communication Manager that retrieves the data to send. In another thread, the Communication Manager is listening for batches sent by other agents and using `PutBatch` to insert them. Therefore, `PutBatch` can be used a maximum of 7 times (6 robots plus the basestation). Taking into account these considerations, at the worst case, the total time taken might be $77.23 \times 7 + 130.58 = 671.19$ microseconds, which is still much below the deadline, giving enough time for other existing logic in the robot.

As curiosity, in figure 5.7, it is possible to find the time consumed by each of the internal operations in `GetBatch` and `PutBatch`. It is visible that most expensive operation is compression or decompression, followed by insertion or retrieval of the batch. Moreover, there are also small operations, such as updating lifetimes or serializing the vector containing the batch, which does not have much impact in the performance done by `GetBatch` or `PutBatch`.



(a) GetBatch

(b) PutBatch

Figure 5.7: The figure contain the time consumed in percentage by each of the internal operations in a `GetBatch` and `PutBatch`. The similar behaviours are marked with the same color for easier comparison. For example, compression and decompression are both marked with green.

Size of the data

The limit for the data size is imposed by the MSL when it states that it does not allow a bandwidth consumption superior to $2.2Mbps$. Therefore, the consumption in bytes per

second is given by $2.2Mbps = 2.2/8 = 0.275MB/s = 270KB/s = 270000B/s$. The unit bytes per second is not relevant in this situation. It is useful to have bytes per cycle of the agent, and that can be calculated by knowing that a robot operates every 20 milliseconds. So, a second has $1000/20 = 50$ cycles and as consequence, the team can send a total of $270000/50 =$ **5400 bytes per cycle**, which is the limit that is going to be considered.

CAMBADA has a total of 7 machines sending data over the network during an official MSL match: 6 robots and 1 basestation. In the RtDB, each robot sends a fixed value of 712 bytes and the basestation sends 396 bytes every cycle. Therefore, the total of bytes sent every cycle is $712 \times 6 + 396 = 4668$ bytes, which is under the limit imposed by MSL, consuming a total of 4668 / 5400 = 86.4% of the available bandwidth.

In figure 5.8, it is possible to verify the bandwidth usage by a robot during four minutes, using RtDB2.

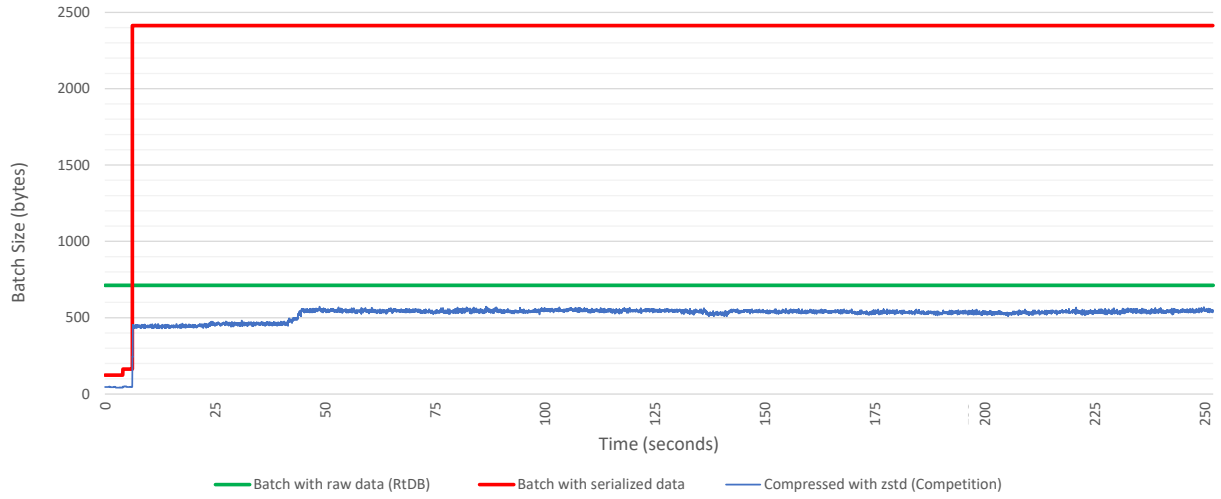


Figure 5.8: Comparison of a robot's data size being sent over the network between RtDB and RtDB2. The chart contains three series: one contains the original data before it gets compressed, another one contains the size of the data after it gets compressed by `zstd` with the dictionary generated in the Portuguese Robotics Open 2017 and the other one is the fixed size sent by RtDB. The horizontal axis represents the time in seconds that ranges from 0 until 4 minutes. The vertical axis corresponds to the size of the data in bytes.

During the first ten seconds approximately occurs the robot's initialization, and during that time the data sent is much smaller, since it is still initializing. The data serialized is around 2400 bytes, but compression is applied when sending data over the network reducing the data a significant amount, as shown in figure 5.8. The maximum value registered during these four minutes was 573 bytes, which is smaller than the RtDB batch size, 712 bytes.

However, it is still important to measure the bandwidth usage by the basestation, and it can be found in figure 5.9. The data serialized by `MsgPack` is smaller than the original size of the data in the RtDB, going from 396 bytes to 200 bytes, without applying any compression. There is already an improvement by only applying serialization, although with compression the maximum batch size is of 59 bytes, which is an improvement of

396 / 59 = 6.7 times smaller than the original size of the data sent by the basestation in the RtDB.

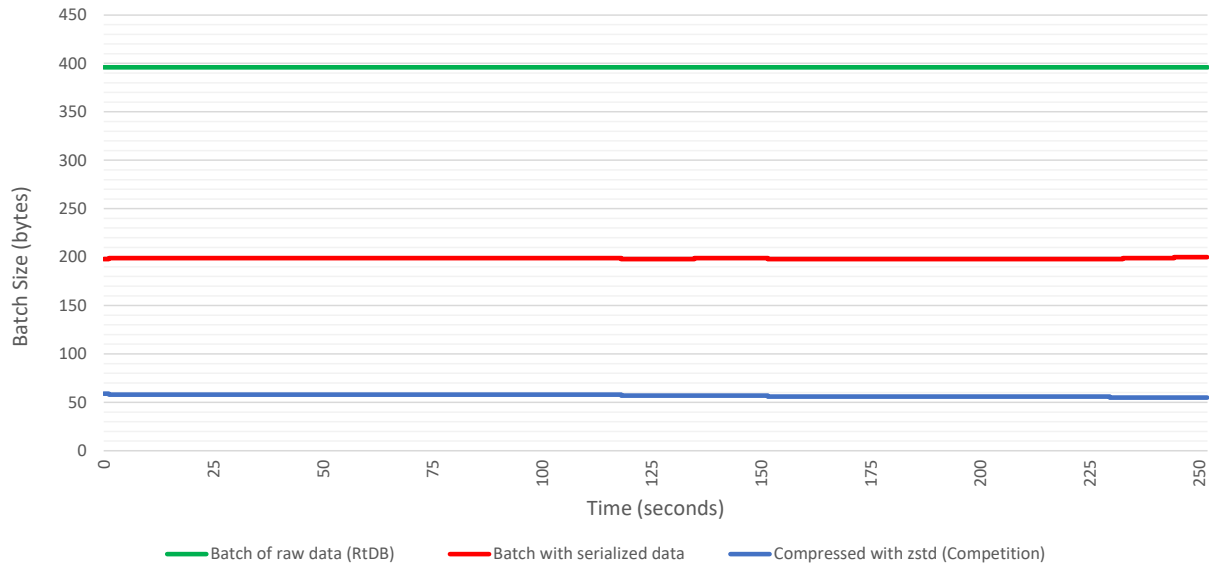


Figure 5.9: Comparison of the basestation's data size being sent over the network between RtDB and RtDB2. The chart contains three series: one contains the original data before it gets compressed, another one contains the size of the data after it gets compressed by `zstd` with the dictionary generated in the Portuguese Robotics Open 2017 and the other one is the fixed size sent by the RtDB. The horizontal axis represents the time in seconds that ranges from 0 until 4 minutes. The vertical axis corresponds to the size of the data in bytes.

To summarize, in this four minutes, the maximum bandwidth that could have been consumed can be calculated by:

$$Bandwidth_{RtDB2} = 573 * 6 + 59 = 3497 \text{ bytes per cycle.}$$

Considering a total of 3497 bytes per cycle, it causes a bandwidth usage of 3497 / 5400 = 64.7%, while RtDB had a fixed bandwidth usage of 86.4%. Therefore, the bandwidth usage in the RtDB2 is one of the improvements achieved, besides all the features described in the implementation.

Chapter 6

Conclusion

An environment with cooperative robots usually has a high demanding characteristics in terms of performance. The robots in the environment need to communicate with each other to cooperate correctly. Typically, they have a high rate of communication, forcing the strategies used to communicate to be efficient. However, this dissertation discussed more than the communication among several agents in a cooperative environment. It also approaches the method used to store all the information received from the network and the internal processes.

The scope of this dissertation comes as an improvement to the previous storage that allowed to obtain a more updated solution with the most recent technologies that could be used in this scenario. The work carried out by this dissertation modifies the way that the system used to communicate and store the information.

On the following section, it is possible to verify how the final results are viable considering the achieved performance and the features that appear with this solution. Later on, there are some mentions about future work that can be done to bring more improvements.

6.1 Validating the results

The dissertation presented had the objective of bringing an updated version of the system used to communicate and store the data among the agents. This updated version had several objectives, briefly described in the chapter 1, such as: allow dynamic data structures to be stored; add some flexibility when sharing the data among several instances; allow structures to be added during the execution of the agent without the storage had had any previously knowledge about the item; improve overall performance and bandwidth; allow to smoothly upgrade the system.

As presented over this dissertation, it was possible to verify that most of these objectives were achieved successfully. The RtDB2 now allows the storage of dynamic data structures, such as vectors from the standard template library from `C++`. The complexity required to add new items to the storage has been removed - it is no longer required to declare an item in the configuration file or to have a synchronized configuration among all instances of the

RtDB2 in order for it to works properly. Moreover, the system was successfully made to be possible to smoothly upgrade from the RtDB to the RtDB2, meaning that it is possible to use the RtDB API with the RtDB2 mechanics through the adapter.

As shown in the results, it was possible to verify that there is a clear trade-off between performance and flexibility when achieving a more complex system in order to obtain more features. There is a noticeable decrease in terms of performance, meaning that the RtDB has a lower response time when accessing its API when compared with the RtDB2. It is an expected trade-off, since the RtDB is a simple model based on a shared memory that is mainly copying memory from an area to another. However, the complexity of the system is transferred to the developer by forcing him to satisfy specific restrictions imposed by the RtDB. In the RtDB2, it became simpler to use the system without having restrictions such as generating a configuration file after adding a new item or having to synchronize that configuration file among all the RtDB2 instances. It was also possible to verify that the bandwidth used can be significantly reduced by the usage of a compressor.

A clear validation of the RtDB2 has been done by using it in two annual competitions of the Middle Size League (MSL) from RoboCup: Portuguese Robotics Open 2017 that took place in Coimbra, Portugal and the international RoboCup 2017 MSL competition that happened in Nagoya, Japan. During the competition, the RtDB2 did not require any adjustments to its implementation, meaning that it was working as expected. It is possible to conclude that the system is viable and has its benefits, when compared with the RtDB.

6.2 Future work

The work carried out by this dissertation left some improvements that can be done to obtain better results and some open research ideas. This dissertation has proven the utility of the RtDB, although some work can still be done in order to improve its result:

- It is possible to store dynamic structures, although most of the structures in CAMBADA are still using preallocated space. It is important to replace all the fields that use preallocated structures and their dimensions should vary with dynamic data structures. It will show benefits in the bandwidth used when transferring the data items over the network and a possible benefit in the performance;

- It is also advisable to completely replace the RtDB with the RtDB2 object in order to reduce the overhead caused by the adapter. Another benefit is that not all the API from the RtDB2 is fully exposed in the adapter, meaning that some features are missing in the adapter, such as obtaining a batch of items including local and shared without compressing them. The main idea of this objective is to completely deprecate any references of the RtDB;

- As referred in section 5.2.2, it might be interesting to study the usage of other serializers in order to increase the performance of the operations `Put` and `Get` in the RtDB2.

However, there will probably be a trade-off in terms of flexibility, since `MsgPack` is one of the most flexible serializers.

Besides the work left to be done related with the usage of the RtDB2 API and features, there is an open idea that can be integrated with the current solution. The actual architecture of CAMBADA uses a module known as Process Manager (PMan) that is responsible for synchronizing the internal processes in a given robot. Most of the processes are awaken only after knowing that a given item is already available; this means that it might be possible to replace PMan with a solution that allows to use the data items as synchronization elements. In this case, it allow the possibility to use the items stored inside the RtDB2 as synchronization elements, thus it would be possible to know when a given item has received a new value or not.

References

[1] RoboCup Objective. `www.robocup.org/objective`. [Online; Accessed December 21, 2017].

[2] Luis Almeida, Frederico Santos, Tullio Facchinetti, Paulo Pedreiras, Valter Silva, and Luís Lopes. Coordinating distributed autonomous agents with a real-time database: The CAMBADA project. *Computer and Information Sciences-ISCIS 2004*, pages 876–886, 2004.

[3] Frederico Miguel do Céu Marques Santos et al. *Architecture for real-time coordination of multiple autonomous mobile units*. PhD thesis, Universidade de Aveiro, 2014.

[4] Barbara Hayes-Roth. A blackboard architecture for control. *Artificial intelligence*, 26(3):251–321, 1985.

[5] Ben Kao and Hector Garcia-Molina. An overview of real-time database systems. In *Real Time Computing*, pages 261–282. Springer, 1994.

[6] Edgar Frank Codd. Derivability, redundancy, and consistency of relations stored in large data banks. Technical report, IBM Research Report, 1969.

[7] Jay Kreibich. *Using SQLite*. " O'Reilly Media, Inc.", 2010.

[8] Theo Haerder and Andreas Reuter. Principles of transaction-oriented database recovery. *ACM Computing Surveys (CSUR)*, 15(4):287–317, 1983.

[9] Rabi Prasad Padhy, Manas Ranjan Patra, and Suresh Chandra Satapathy. RDBMS to NoSQL: Reviewing some next-generation non-relational databases. *International Journal of Advanced Engineering Science and Technologies*, 11(1):15–30, 2011.

[10] Yu Huang and Tie-jian Luo. NoSQL Database: A Scalable, Availability, High Performance Storage for Big Data. In *Joint International Conference on Pervasive Computing and the Networked World*, pages 172–183. Springer, 2013.

[11] ABM Moniruzzaman and Syed Akhter Hossain. NoSQL database: New era of databases for big data analytics-classification, characteristics and comparison. *arXiv preprint arXiv:1307.0191*, 2013.

[12] Katarina Grolinger, Wilson A Higashino, Abhinav Tiwari, and Miriam AM Capretz. Data management in cloud environments: NoSQL and NewSQL data stores. *Journal of Cloud Computing: Advances, Systems and Applications*, 2(1):22, 2013.

[13] Dan Pritchett. Base: An ACID alternative. *Queue*, 6(3):48–55, 2008.

[14] Eric A Brewer. Towards robust distributed systems. In *PODC*, volume 7, 2000.

[15] Hector Garcia-Molina and Kenneth Salem. Main memory database systems: An overview. *IEEE Transactions on knowledge and data engineering*, 4(6):509–516, 1992.

[16] Christof Strauch, Ultra-Large Scale Sites, and Walter Kriha. NoSQL databases. *Lecture Notes, Stuttgart Media University*, 2011.

[17] Yishan Li and Sathiamoorthy Manoharan. A performance comparison of SQL and NoSQL databases. In *Communications, Computers and Signal Processing (PACRIM), 2013 IEEE Pacific Rim Conference on*, pages 15–19. IEEE, 2013.

[18] Neal Leavitt. Will NoSQL databases live up to their promise? *Computer*, 43(2), 2010.

[19] Rick Cattell. Scalable SQL and NoSQL data stores. *Acm Sigmod Record*, 39(4):12–27, 2011.

[20] Michael A Olson, Keith Bostic, and Margo I Seltzer. Berkeley DB. In *USENIX Annual Technical Conference, FREENIX Track*, pages 183–191, 1999.

[21] Robert Love. *Linux System Programming: Talking Directly to the Kernel and C Library*. " O'Reilly Media, Inc.", 2013.

[22] Russell Sears and Eric Brewer. Segment-based recovery: write-ahead logging revisited. *Proceedings of the VLDB Endowment*, 2(1):490–501, 2009.

[23] Howard Chu. Mdb: A memory-mapped database and backend for OpenLDAP. In *Proceedings of the 3rd International Conference on LDAP, Heidelberg, Germany*, page 35, 2011.

[24] Matthew Hardin. Understanding LMDB Database File Sizes and Memory Utilization. `https://symas.com/understanding-lmdb-database-file-sizes-and-memory-utilization/`, 2016. [Online; Accessed September 12, 2017].

[25] David Bell and Jane Grimson. *Distributed database systems*. Addison-Wesley Longman Publishing Co., Inc., 1992.

[26] M Tamer Özsu and Patrick Valduriez. *Principles of distributed database systems*. Springer Science & Business Media, 2011.

[27] Tiago Macedo and Fred Oliveira. *Redis Cookbook*. " O'Reilly Media, Inc.", 2011.

[28] Josiah L Carlson. *Redis in Action*. Manning Publications Co., 2013.

[29] Debra A Lelewer and Daniel S Hirschberg. Data compression. *ACM Computing Surveys (CSUR)*, 19(3):261–296, 1987.

[30] Khalid Sayood. *Introduction to data compression*. Newnes, 2012.

[31] Jacob Ziv and Abraham Lempel. Compression of individual sequences via variable-rate coding. *IEEE transactions on Information Theory*, 24(5):530–536, 1978.

[32] Victor R Lesser and Lee D Erman. A Retrospective View of the Hearsay-II Architecture. In *IJCAI*, volume 5, pages 790–800, 1977.

[33] Daniel D. Corkill. Blackboard and Multi-Agent Systems & the Future. *In Proceedings of the International Lisp Conference, New York, USA*, 2003.

[34] R. Dias, F. Amaral, J.L. Azevedo, M. Azevedo, B. Cunha, P. Dias, N Lau, A. J. R. Neves, E. Pedrosa, A. Pereira, J. Silva, and A. Trifan. CAMBADA'2016: Team Description Paper 2016. *Proceedings Robocup 2016*, 2016.

Appendices

Appendix A

RtDB Configuration

The following examples in this chapter do not contain the complete configuration from CAMBADA. There are a total of three files, one that is manually configured and two automatically generated.
The first file, that is manually configured, is the one shown in listing A.1. It has a custom format created for this specific purpose and it is composed of four different definitions:

1. A list of agents that simply requires the name of each agent;
2. A list of items. Each item from the list is characterized by three fields:
   (a) An item identifier that is a simple string without any spaces and must be unique;
   (b) The name of the datatype. This is the name of the structure or the class that is going to be stored;
   (c) The location of the datatype, which will probably be inside a header file.
3. A list of possible schemas, which has the purpose of marking a specific item as shared or local. A schema may contain several items or even all of them;
4. A list of assignments that corresponds to direct assignments of a schema to one or more agents. It allows two agents to behave differently with the same items, one agent can share an item that the other one is keeping as local.

Listing A.1: File used to configure RtDB internal storage (rtdb.conf)

```
# Configuration file for RtDB items.
# - It is composed of 3 sections: agents, items, and schemas.
#   - The agents section is a comma-separated list of agent's ids.
#   - The items section is a list of items.
#     - An item is composed of an id, a datatype, the headerfile where
#       the datatype is declared, and a period.
#     - The item format is described bellow.
```

```
#   - The schema section is a newline- or semicolon-separated list of
    schemas.
#     - A schema is an arrangement of items,
#       each one labelled as either shared or local.
#     - A schema must also be assigned to 1 or more agents.
#     - The schema formar is described bellow.
# - Everything from # to end of line is discarded.
# - Empty lines are also discarded.
# - In the comments bellow square brackets define an optional component.

# Agent declaration section
#   it is a comma-separated list of agent's ids
#
# AGENTS = «id» [, «id» , ...] [;]
#
AGENTS = BASE_STATION, CAMBADA_1, CAMBADA_2, CAMBADA_3;

# Item declaration section
#
# ITEM «id» { datatype = «id»; [headerfile = «filename»];
# [period = «number»]; }
# headerfile defaults to «datatype» plus ".h". For instance if datatype
   = abc,
#   then headerfile defaults to abc.h
# period defaults to 1
#
ITEM ROBOT_WS { datatype = Robot; headerfile = Robot.h; }
ITEM LAPTOP_INFO { datatype = LaptopInfo; headerfile = SystemInfo.h; }
ITEM COACH_INFO { datatype = CoachInfo; headerfile = CoachInfo.h; }
ITEM VISION_INFO { datatype = VisionInfo; headerfile = VisionInfo.h; }
ITEM FORMATION_INFO { datatype = FormationInfo; headerfile =
   CoachInfo.h; }
ITEM GRIDVIEW { datatype = GridView; headerfile = GridView.h; }
ITEM COACHMAP { datatype = CoachMap; headerfile = CoachLogModeInfo.h; }
ITEM MONITOR_RTDB { datatype = MonitorRTDB; headerfile = monitor_rtdb.h;
   }
ITEM TEAMPLAY { datatype = TeamPlayRtdb; headerfile = TeamPlay.h; }

# SCHEMA definition section
#
# SCHEMA «id» { [shared = «id» [ , «id», ...] ; ]
#    [local = «id» [, «id», ...] ; ]
#
SCHEMA BaseStation
{
    shared = COACH_INFO, FORMATION_INFO;
    local = GRIDVIEW, COACHMAP, TEAMPLAY;
}

SCHEMA Player
{
```

```
    shared = ROBOT_WS, LAPTOP_INFO, MONITOR_RTDB;
    local = COACH_INFO, VISION_INFO;
}

# ASSIGNMENT definition section
#
# ASSIGNMENT { schema = «id»; agents = «id», ... ; }
#
ASSIGNMENT { schema = BaseStation; agents = BASE_STATION; }
ASSIGNMENT { schema = Player;  agents = CAMBADA_1, CAMBADA_2, CAMBADA_3;
   }
```

The following two listings are generated by `xrtdb` when fed by the previous listing (`rtdb.conf`):

1. `rtdb.ini` (as shown in listing A.2) that is a file loaded by the RtDB on its startup. Its purpose is to store the specific item in the correct location in the shared memory, by using the bytes from the previous items and their offsets. It also works for Communication Manager to decide if an item is sent over the network or not;

Listing A.2: File generated by xrtdb (rtdb.ini) where the RtDB reads its configuration

```
##
## rtdb.ini IS AN AUTOGEN FILE, DO NOT EDIT.
##


# 0    BASE_STATION
2    260       1    s
4    88        1    s
5    60004     1    l
6    2         1    l
8    36        1    l

# 1    CAMBADA_1
0    608       1    s
1    2         1    s
7    6         1    s
2    260       1    l
3    9052      1    l

# 2    CAMBADA_2
0    608       1    s
1    2         1    s
7    6         1    s
2    260       1    l
3    9052      1    l

# 3    CAMBADA_3
0    608       1    s
1    2         1    s
```

```
7     6          1    s
2     260        1    l
3     9052       1    l
```

2. `rtdb_user.h` (as shown in listing A.3) is an header file that contains a list of constant values defined using the preprocessor `define`, macros. These definitions allow the programmer to insert or retrieve data from the RtDB using user-friendly macros without having to think where the item is stored or not. The item can easily be accessed by the name set in the agents from the `rtdb.conf`, and the same goes for the name of each item.

Listing A.3: Header file (rtdb_user.h) with macros for the RtDB data items and agents

```
/* AUTOGEN FILE : rtdb_user.h */

#ifndef _CAMBADA_RTDB_USER_
#define _CAMBADA_RTDB_USER_

/* agents section */

#define BASE_STATION  0
#define CAMBADA_1 1
#define CAMBADA_2 2
#define CAMBADA_3 3

#define N_AGENTS  4

/* items section */

#define ROBOT_WS  0
#define LAPTOP_INFO 1
#define COACH_INFO  2
#define VISION_INFO 3
#define FORMATION_INFO  4
#define GRIDVIEW  5
#define COACHMAP  6
#define MONITOR_RTDB  7
#define TEAMPLAY  8

#define N_ITEMS 9

#endif

/* EOF : rtdb_user.h */
```

Appendix B

Compression Comparison

This appendix contains a table resultant from the tool `lzbench`. It is the complete table that resulted from the benchmark used in section 4.4.3. The `lzbench` was executed under the hardware specified in section 5.1, which means that it was done under an Ubuntu 14.04 and an Intel Core i5-3340M CPU @ 2.7 GHz. All the tests were done over the same data structure. This data was obtained by compressing a batch of serialized data resultant from `MsgPack`.

Table B.1: Comparison among existing compressors resultant from `lzbench`. All the values mentioned in this table are in megabytes per second, with the exception of the data size that is expressed in bytes and ratio in percentage. In the table, there is a separation between the compression and decompression speeds obtained. In the speeds it is possible to find out the maximum speed obtained, the average and the median value.

	Compression (MB/s)			Decompression (MB/s)			Data Size (bytes)		
Compressor name	**Maximum**	**Average**	**Median**	**Maximum**	**Average**	**Median**	**Original**	**Compress**	**Ratio**
memcpy	9661.16	139.92	197.37	9906.78	447.29	445.59	2338	2338	100
blosclz 2015-11-10 -1	1678.39	193.37	200.98	10034.33	1192.25	9906.78	2338	2338	100
blosclz 2015-11-10 -3	952.34	180.96	189.83	9991.45	627.31	9906.78	2338	2338	100
blosclz 2015-11-10 -6	358.26	174.62	193.78	1647.64	269.85	214.4	2338	1342	57.4
blosclz 2015-11-10 -9	358.59	172.8	191.09	1647.64	264.57	210.31	2338	1342	57.4
brieflz 1.1.0	35.42	34.77	35.01	303.09	163.28	204.1	2338	1349	57.7
brotli 2017-03-10 -0	122.74	120.6	120.71	178.02	175	174.71	2338	1339	57.27
brotli 2017-03-10 -2	70.02	68.51	68.53	161.78	158.93	158.77	2338	1231	52.65
brotli 2017-03-10 -5	21.44	20.81	20.93	207.58	202.44	203	2338	1055	45.12
brotli 2017-03-10 -8	17.89	17.39	17.54	207.05	202.85	203.27	2338	1055	45.12
brotli 2017-03-10 -11	0.41	0.4	0.41	156.72	153.14	153.18	2338	986	42.17
crush 1.0 -0	0.3	0.3	0.3	295.84	171.85	182.94	2338	1275	54.53
crush 1.0 -1	0.31	0.3	0.3	299.59	168.91	207.47	2338	1270	54.32
crush 1.0 -2	0.3	0.3	0.3	296.63	172.83	208.1	2338	1269	54.28
csc 2016-10-13 -1	9.05	8.88	8.94	27.88	27.11	27.23	2338	1156	49.44
csc 2016-10-13 -3	5.98	5.86	5.87	27.66	26.86	26.96	2338	1157	49.49
csc 2016-10-13 -5	3.31	3.28	3.29	27.99	27.19	27.39	2338	1152	49.27
density 0.12.5 beta -1	103.06	100.91	101.75	97.66	93.61	94.55	2338	2032	86.91
density 0.12.5 beta -2	28.16	27.48	27.66	29.16	28.04	28.03	2338	1802	77.07

	Compression (MB/s)			Decompression (MB/s)			Data Size (bytes)		
Compressor name	**Maximum**	**Average**	**Median**	**Maximum**	**Average**	**Median**	**Original**	**Compress**	**Ratio**
density 0.12.5 beta -3	9.92	9.42	9.38	9.4	9.13	9.12	2338	1756	75.11
fastlz 0.1 -1	252.35	224.29	226.35	1115.99	205.54	202.06	2338	1342	57.4
fastlz 0.1 -2	260.59	189.24	224.05	1025.44	204.85	196.88	2338	1342	57.4
gipfeli 2016-07-13	269.29	220.23	216.72	360.69	146.46	209.44	2338	1306	55.86
libdeflate 0.7 -1	37.95	37.43	37.75	233.8	222.9	227.99	2338	1163	49.74
libdeflate 0.7 -3	38.08	37.56	37.72	233.8	223.73	229.31	2338	1159	49.57
libdeflate 0.7 -6	36.9	36.38	36.51	233.8	223.9	229.15	2338	1156	49.44
libdeflate 0.7 -9	15.86	15.48	15.66	233.8	223.84	229.06	2338	1155	49.4
libdeflate 0.7 -12	7.86	7.7	7.75	233.8	223.86	228.97	2338	1152	49.27
lizard 1.0 -10	510.7	141.75	163.58	3630.43	241.16	218.12	2338	1361	58.21
lizard 1.0 -12	151.48	147.56	148.57	3537.06	228.41	214.97	2338	1349	57.7
lizard 1.0 -15	98.28	86.75	95.52	3564.02	223.09	217.08	2338	1337	57.19
lizard 1.0 -19	1.01	1	1	3413.14	228.86	215.86	2338	1328	56.8
lizard 1.0 -20	444.06	137.89	144.95	2690.45	217.63	212.53	2338	1406	60.14
lizard 1.0 -22	186.1	182.91	183.89	2603.56	223.82	215.6	2338	1319	56.42
lizard 1.0 -25	1.05	1.05	1.04	2074.53	216.32	212.55	2338	1248	53.38
lizard 1.0 -29	0.95	0.96	0.95	2072.7	210.16	209.39	2338	1241	53.08
lizard 1.0 -30	488.81	129.25	175.41	3630.43	226.92	214.12	2338	1361	58.21
lizard 1.0 -32	160.86	154.74	156.11	3531.72	242.28	213.63	2338	1349	57.7
lizard 1.0 -35	98.13	86.62	95.5	3542.42	217.45	214	2338	1342	57.4
lizard 1.0 -39	1	0.99	1	3413.14	234.6	216.4	2338	1328	56.8
lizard 1.0 -40	278.27	176.47	196.97	2690.45	224.48	211.43	2338	1406	60.14
lizard 1.0 -42	154.85	151.97	152.72	2597.78	221.53	213.2	2338	1319	56.42
lizard 1.0 -45	1.04	1.05	1.04	2074.53	208.1	209.09	2338	1248	53.38
lizard 1.0 -49	0.94	0.95	0.94	1652.3	203.25	207.12	2338	1218	52.1
lz4 1.7.5	569.55	169.84	168.57	3238.23	310.08	220.69	2338	1326	56.72
lz4fast 1.7.5 -3	862.09	183.52	183.92	3569.47	344.53	228.36	2338	1466	62.7
lz4fast 1.7.5 -17	1615.76	234.01	169.97	4531.01	355.48	226.02	2338	1853	79.26
lz4hc 1.7.5 -1	80.64	78.81	79.26	3283.71	345.55	224.05	2338	1318	56.37
lz4hc 1.7.5 -4	79.76	77.83	78.3	3292.96	316.12	221.32	2338	1309	55.99
lz4hc 1.7.5 -9	79.17	77.19	77.63	3292.96	356.57	221.13	2338	1308	55.95
lz4hc 1.7.5 -12	15.4	15.18	15.25	3354.38	337.52	224.74	2338	1308	55.95
lzf 3.6 -0	153.98	149.74	153.66	955.85	201.24	190.51	2338	1338	57.23
lzf 3.6 -1	151.67	146.82	151.1	925.21	198.51	186.8	2338	1339	57.27
lzfse 2017-03-08	33.77	33.35	33.64	649.99	167.98	169.04	2338	1286	55
lzg 1.0.8 -1	0.29	0.28	0.28	588.62	177.01	171.31	2338	1321	56.5
lzg 1.0.8 -4	0.29	0.28	0.28	589.96	177.85	172.89	2338	1320	56.46
lzg 1.0.8 -6	0.28	0.28	0.28	589.81	178.98	172.57	2338	1320	56.46
lzg 1.0.8 -8	0.28	0.27	0.27	590.11	175.47	173.11	2338	1320	56.46
lzjb 2010	358.92	197.02	186.94	709.13	223.56	180.19	2338	1369	58.55
lzlib 1.8 -0	3.56	3.39	3.53	33.23	30.73	31.95	2338	1093	46.75
lzlib 1.8 -3	4.1	4.03	4.05	33.15	31.56	32.04	2338	1096	46.88
lzlib 1.8 -6	3.04	2.98	3	33.28	31.69	32.1	2338	1094	46.79
lzlib 1.8 -9	2.29	2.25	2.26	33.38	31.71	32.08	2338	1094	46.79
lzma 16.04 -0	7	6.82	6.95	49.49	47.35	47.78	2338	1102	47.13
lzma 16.04 -2	5.89	5.7	5.81	49.44	47.4	47.77	2338	1102	47.13
lzma 16.04 -4	1.19	0.88	0.9	49.57	47.41	47.7	2338	1102	47.13
lzma 16.04 -5	0.46	0.45	0.45	51.39	49.31	49.73	2338	1068	45.68
lzma 16.04 -9	276.75	124.68	130.04	10034.33	645.5	9906.78	2338	2338	100
lzo1 2.09 -1	313.83	182.86	185.79	1027.69	202.58	193.98	2338	1344	57.49
lzo1 2.09 -99	49.81	48.56	48.64	1018.74	212.28	198.57	2338	1337	57.19
lzo1a 2.09 -1	315.52	183.42	184.85	1346	217.33	207.8	2338	1326	56.72
lzo1a 2.09 -99	49.58	48.22	48.26	1334.47	209.09	208.73	2338	1318	56.37
lzo1b 2.09 -1	231.42	221.17	227.92	1639.55	249.87	212.47	2338	1348	57.66
lzo1b 2.09 -3	223.56	212.55	219.55	1636.11	247.49	208.6	2338	1348	57.66
lzo1b 2.09 -6	208.68	206.08	206.5	1508.39	222.31	207.49	2338	1336	57.14
lzo1b 2.09 -9	103.24	100.27	100.9	1513.27	227.9	210.84	2338	1332	56.97

	Compression (MB/s)			Decompression (MB/s)			Data Size (bytes)		
Compressor name	**Maximum**	**Average**	**Median**	**Maximum**	**Average**	**Median**	**Original**	**Compress**	**Ratio**
lzo1b 2.09 -99	51.02	50.54	50.5	1520.16	225.5	211.79	2338	1323	56.59
lzo1b 2.09 -999	65.9	63.81	64.62	1503.54	224.35	211.18	2338	1316	56.29
lzo1c 2.09 -1	334.91	175.6	196.32	1880.93	252.24	208.73	2338	1339	57.27
lzo1c 2.09 -3	319.4	169.52	187.41	1879.42	247.04	213.69	2338	1339	57.27
lzo1c 2.09 -6	337.03	162.43	184.91	1706.57	226.66	214.46	2338	1326	56.72
lzo1c 2.09 -9	128.61	123.83	125.76	1715.33	234.9	218.61	2338	1322	56.54
lzo1c 2.09 -99	77.81	75.65	75.7	1731.85	227.54	217.43	2338	1313	56.16
lzo1c 2.09 -999	65.77	64.76	64.94	1712.82	226.46	215.56	2338	1305	55.82
lzo1f 2.09 -1	342.61	177.93	197.73	1556.59	240.07	206.37	2338	1337	57.19
lzo1f 2.09 -999	64.31	63.34	63.32	1441.43	216.04	209.27	2338	1293	55.3
lzo1x 2.09 -1	424.55	155.83	208.96	2287.67	283.29	216.94	2338	1380	59.02
lzo1x 2.09 -11	538.34	164.17	170.76	2294.41	275.35	215.9	2338	1384	59.2
lzo1x 2.09 -12	523.28	170.18	193.9	2294.41	279.73	216.38	2338	1381	59.07
lzo1x 2.09 -15	503.66	168.71	199.54	2294.41	278.2	217.49	2338	1381	59.07
lzo1x 2.09 -999	17.8	17.58	17.64	1532.11	207.42	203.07	2338	1271	54.36
lzo1y 2.09 -1	427.03	155.74	204.89	1818.04	266.38	207.97	2338	1351	57.78
lzo1y 2.09 -999	18.6	18.37	18.5	1276.9	208.6	201	2338	1242	53.12
lzo1z 2.09 -999	17.45	17.16	17.24	1470.44	216.48	201.26	2338	1259	53.85
lzo2a 2.09 -999	47.83	46.83	46.44	914.71	182.29	190.97	2338	1250	53.46
lzrw 15-Jul-1991 -1	346.47	220.59	221.34	581.45	178.47	172.25	2338	1388	59.37
lzrw 15-Jul-1991 -3	311.86	167.9	190.05	634.98	176.32	175.01	2338	1388	59.37
lzrw 15-Jul-1991 -4	295.39	135.58	166.92	314.92	165.52	196.4	2338	1388	59.37
lzrw 15-Jul-1991 -5	109.45	107.41	107.87	294.64	186.76	204.51	2338	1372	58.68
lzsse2 2016-05-14 -1	5.65	5.44	5.51	995.32	195.89	185.63	2338	1505	64.37
lzsse2 2016-05-14 -6	5.11	4.93	4.98	1004.73	190.42	186.09	2338	1478	63.22
lzsse2 2016-05-14 -12	4.87	4.91	4.94	1004.73	194.2	181.35	2338	1478	63.22
lzsse2 2016-05-14 -16	4.61	4.92	4.98	1004.73	198.62	184.85	2338	1478	63.22
lzsse4 2016-05-14 -1	4.83	5.53	4.66	1576.53	237.94	208.27	2338	1417	60.61
lzsse4 2016-05-14 -6	5.25	4.63	4.37	1598.09	229.49	204.66	2338	1407	60.18
lzsse4 2016-05-14 -12	5.08	5.08	4.42	1596.99	241.23	207.32	2338	1407	60.18
lzsse4 2016-05-14 -16	4.51	5.08	4.41	1598.09	231.78	209.07	2338	1407	60.18
lzsse8 2016-05-14 -1	5.63	5.42	5.49	1903.91	235.47	209.01	2338	1389	59.41
lzsse8 2016-05-14 -6	4.98	4.96	5.01	1902.36	251.61	214.22	2338	1379	58.98
lzsse8 2016-05-14 -12	4.4	4.51	4.59	1900.81	263.59	212.55	2338	1379	58.98
lzsse8 2016-05-14 -16	5.14	4.26	4.3	1902.36	255.02	211.93	2338	1379	58.98
lzvn 2017-03-08	33.83	33.53	33.77	1200.82	206.99	200.22	2338	1270	54.32
pithy 2011-12-24 -0	451.87	159.02	194.27	2747.36	293.13	217.87	2338	1430	61.16
pithy 2011-12-24 -3	431.05	161.21	206.79	2753.83	286.2	221.93	2338	1424	60.91
pithy 2011-12-24 -6	426.33	159.5	215.72	2744.13	302.81	218.83	2338	1423	60.86
pithy 2011-12-24 -9	410.9	185.88	169.9	2744.13	286.66	225.87	2338	1423	60.86
quicklz 1.5.0 -1	74.78	72.11	71.87	241.2	215.4	213.65	2338	1351	57.78
quicklz 1.5.0 -2	69.3	67.04	67	217.37	214.18	215.46	2338	1361	58.21
quicklz 1.5.0 -3	52.44	51.22	51.39	551.03	151.64	157.54	2338	1329	56.84
slz_zlib 1.0.0 -1	198.17	194.19	195.6	249.6	179.18	217.71	2338	1291	55.22
slz_zlib 1.0.0 -2	198.07	194.32	195.26	250.37	191.4	217.45	2338	1291	55.22
slz_zlib 1.0.0 -3	198.22	194.09	195.45	249.81	186.52	217.43	2338	1291	55.22
snappy 1.1.4	789.07	158.49	173.7	3036.36	277.57	216.44	2338	1445	61.8
tornado 0.6a -1	177.98	174.17	174.46	355.16	149.34	197.2	2338	2022	86.48
tornado 0.6a -2	104.14	102.63	102.48	359.91	140.82	203.46	2338	1810	77.42
tornado 0.6a -3	33.34	32.93	32.99	54.06	51.96	53.09	2338	1739	74.38
tornado 0.6a -4	26.41	26.13	26.26	53.97	51.96	53.12	2338	1738	74.34
tornado 0.6a -5	14.75	14.58	14.58	60.17	59.5	59.66	2338	1792	76.65
tornado 0.6a -6	13.26	13.05	13.08	60.29	59.52	59.72	2338	1790	76.56
tornado 0.6a -7	7.96	7.87	7.9	60.19	59.6	59.56	2338	1788	76.48
tornado 0.6a -10	5.61	5.53	5.57	60.03	59.29	59.48	2338	1785	76.35
tornado 0.6a -13	5.21	5.14	5.15	77.23	76.26	76.5	2338	1282	54.83
tornado 0.6a -16	4.93	3.87	3.89	77.28	76.17	76.43	2338	1279	54.7
ucl_nrv2b 1.03 -1	31.29	30.65	30.72	501.07	151.35	169.52	2338	1189	50.86
ucl_nrv2b 1.03 -6	28.55	27.92	28.04	511.15	149.63	163.81	2338	1177	50.34
ucl_nrv2b 1.03 -9	26.45	25.95	26.09	507.93	150.26	162.07	2338	1177	50.34

	Compression (MB/s)			Decompression (MB/s)			Data Size (bytes)		
Compressor name	**Maximum**	**Average**	**Median**	**Maximum**	**Average**	**Median**	**Original**	**Compress**	**Ratio**
ucl_nrv2d 1.03 -1	30.98	30.36	30.52	456.11	149.53	158.78	2338	1178	50.38
ucl_nrv2d 1.03 -6	27.74	27.19	27.3	457.98	147.86	162.09	2338	1170	50.04
ucl_nrv2d 1.03 -9	25.83	25.31	25.42	465.37	150.2	160.62	2338	1170	50.04
ucl_nrv2e 1.03 -1	31.11	30.54	30.67	405.34	143.89	157.21	2338	1177	50.34
ucl_nrv2e 1.03 -6	28.21	27.7	27.82	407.81	149.06	165.67	2338	1169	50
ucl_nrv2e 1.03 -9	26.68	26.06	26.15	409.67	145.32	156.46	2338	1169	50
xpack 2016-06-02 -1	55.13	54.12	54.41	366.8	152.38	207.18	2338	1228	52.52
xpack 2016-06-02 -6	52.12	51.12	51.44	370.58	165.97	210.14	2338	1222	52.27
xpack 2016-06-02 -9	52.24	51.13	51.42	370.46	155.23	192.4	2338	1222	52.27
xz 5.2.3 -0	8.11	7.97	8.03	38.44	36.96	37.19	2338	1114	47.65
xz 5.2.3 -3	0.84	0.82	0.7	35.89	34.62	34.93	2338	1115	47.69
xz 5.2.3 -6	0.37	0.34	0.34	36.57	35.09	35.33	2338	1082	46.28
xz 5.2.3 -9	141.19	138.31	21.28	10034.33	688.25	9906.78	2338	2338	100
yalz77 2015-09-19 -1	16.22	6.33	6.52	1135.5	179.75	188.84	2338	1351	57.78
yalz77 2015-09-19 -4	2.34	2.21	2.25	1151.16	177.46	187.7	2338	1336	57.14
yalz77 2015-09-19 -8	1.15	1.09	1.06	1127.29	179.42	190.03	2338	1333	57.01
yalz77 2015-09-19 -12	0.77	0.73	0.72	1126.2	177.79	189.02	2338	1333	57.01
yappy 2014-03-22 -1	163.1	158.15	158.04	4175	341.86	222.24	2338	1319	56.42
yappy 2014-03-22 -10	123.9	121.26	121.61	4182.47	361.81	228.79	2338	1309	55.99
yappy 2014-03-22 -100	115.7	113.53	113.81	4182.47	352	230.07	2338	1309	55.99
zlib 1.2.11 -1	51.66	49.52	49.99	195.65	189.07	189.25	2338	1171	50.09
zlib 1.2.11 -6	40.12	38.53	38.93	196.9	190.44	191.23	2338	1163	49.74
zlib 1.2.11 -9	39.49	37.87	38.4	195.99	190.67	191.28	2338	1163	49.74
zling 2016-01-10 -0	0.46	0.45	0.45	1.18	1.13	1.12	2338	1430	61.16
zling 2016-01-10 -1	0.46	0.45	0.45	1.21	1.14	1.12	2338	1428	61.08
zling 2016-01-10 -2	0.46	0.45	0.45	1.19	1.13	1.12	2338	1427	61.04
zling 2016-01-10 -3	0.46	0.45	0.45	1.22	1.14	1.12	2338	1427	61.04
zling 2016-01-10 -4	0.46	0.45	0.45	1.22	1.14	1.12	2338	1427	61.04
zstd 1.3.1 -1	120.55	118.18	119.27	346.73	148.8	198.46	2338	1199	51.28
zstd 1.3.1 -2	117.72	115.89	117.44	289.57	160.03	208.69	2338	1165	49.83
zstd 1.3.1 -5	71.6	70.21	70.78	292.91	156.4	207.12	2338	1163	49.74
zstd 1.3.1 -8	63.18	61.94	62.34	292.65	152.68	205.56	2338	1163	49.74
zstd 1.3.1 -11	16.15	15.83	15.86	277.61	173.11	218.06	2338	1138	48.67
zstd 1.3.1 -15	4.54	4.48	4.5	281.42	165.31	198.12	2338	1132	48.42
zstd 1.3.1 -18	4.53	4.48	4.5	281.38	164.38	206.17	2338	1132	48.42
zstd 1.3.1 -22	4.93	4.87	4.88	281.99	162.54	207.82	2338	1132	48.42
shrinker 0.1	496.71	129.32	158.4	2461.05	320.32	218.4	2338	1305	55.82
wflz 2015-09-16	10.63	10.57	10.6	1493.93	264.45	209.22	2338	1395	59.67
lzmat 1.01	92.59	90.66	92.07	444.91	158.53	157.12	2338	1260	53.89